Lab 6. Developing with Alternative Tools: Buildah, Podman, Skopeo

The objective of this lab exercise is to get you started with alternative tools to run containers, build images and to work with registries. In this lab, you are going to learn:

- How to run containers directly with runc and containerd, and how those tools differ from Docker CLI.
- How to run containers and build images in a rootless and daemonless environment with Podman and Buildah.
- How to build container images from scratch, as well as how to do it step by step.
- How to inspect images with Skopeo.

Launching Containers with runc

To launch a container with runc, do:

```
runc
runc spec
cat config.json
```

Now, create a root file system by exporting a container and launch it with runc:

```
docker create --name alp01 alpine sh
docker export alp01 -o alpine-root.tar
mkdir rootfs
tar -xf alpine-root.tar -C ./rootfs/
runc run testc
```

Inside the container:

```
ps aux
```

```
ifconfig
cat /etc/issue
touch test
```

While the container is running, open a second terminal and examine it with the following command:

```
runc list
runc exec -t testc sh
```

***NOTE**: Run `^d` or `exit` command to exit the container shell.*

```
ctr run -t --net-host --rm docker.io/library/alpine:3.12
demo sh
```

Working with containerd

In addition to launching containers with runc, you can also run it using containerd. Containerd runs as a daemon, does image management, sets up the root file system, and then calls runc to finally launch the container. To work with containerd, try using the following sequence of commands, and follow along with the video lesson:

```
ctr
```

```
ctr version
```

```
ctr container
```

```
ctr image
```

```
ctr image pull nginx
ctr image pull docker.io/library/nginx
ctr image pull docker.io/library/nginx:latest
```

```
ctr run -h
```

```
ctr image pull docker.io/library/alpine:latest
ctr run -t --rm docker.io/library/alpine:latest sh
```

Inside the container:

```
ps
cat /etc/issue
ifconfig
exit
```

Outside the container:

```
ctr run -h
ctr run -t --net-host --rm docker.io/library/alpine:latest sh
```

Inside the container:

```
ps aux
ifconfig
exit
```

Outside the container:

```
ctr run -t -d docker.io/library/nginx:latest
web ctr c list
```

```
ctr c list
ctr c rm web
ctr t
ctr t ls
ctr t kill web
ctr c list
ctr c rm web
```

Using Podman as a Replacement for Docker

While containerd and runc are underlying technologies that Docker uses, if there is one tool that you can replace Docker with, it's Podman. The resemblance is so strong that you can create an alias (e.g. **`alias docker=podman`**), and you may not even notice the difference. All the common commands that come with Docker are supported by Podman. There are distinct advantages of using Podman such as ability to run containers as a non-root user, as well as ability to run containers without a daemon, which make Podman more secure and flexible. In this section, you are going to start exploring Podman.

Daemonless Alternative to Docker

Pulling an image:

```
podman image pull mysql
```

Launching a container:

```
podman run -idt -P nginx
```

Building an image:

```
git clone xxxx
cd xxxx
```

```
podman image build -t xxxx/xxxx:xx .
```

Rootless Containers with Podman

Create a new user, switch to it and start running containers with that user. This is not possible with Docker unless you add that user to the Docker group, which essentially provides the user with root access (Docker daemon always runs as root).

```
useradd -m devops
passwd devops

su - devops

podman image ls
podman run -idtP nginx

podman ps
podman image ls

podman run -idt -p 80:80 nginx
```

Working with Pods

Podman also supports running a group of containers with shared namespaces, commonly called pods as per the Kubernetes terminology. This makes it easier to develop with pods, without requiring you to set up a Kubernetes environment.

```
podman ps
podman pod ls

podman pod create --name web -p 80

podman run -idt --pod web --name nginx nginx:stable-alpine

podman run -idt --pod web --name sync schoolofdevops/sync:v2

podman pod ls podman
pod ls --ctr-name
podman ps
podman ps --pod

podman generate kube web
podman generate kube -s web
```

Advanced Image Building with Buildah

Building an Image with Dockerfile as a Non-root User

Similar to Podman, if you want to build images as a non-root user, and in a daemonless mode, you have a companion to Podman called Buildah. Install Buildah and try building an image with it using the same spec that you would use with Docker, e.g. with Dockerfile. Follow the instructions below and refer to the video lessons:

```
sudo apt-get -y install buildah
```

```
buildah
buildah bud -h
```

```
git clone
https://github.com/schoolofdevops/example-voting-app.git
```

```
cd example-voting-app/vote/
```

```
MYACCOUNT=xxxxxx
```

***NOTE**: Replace **xxxxxx** with your username/organization/project on Docker Hub to push the images.*

```
buildah bud -t docker.io/$MYACCOUNT/vote:v1 .
```

```
buildah images
podman image history docker.io/$MYACCOUNT/vote:v1
```

Building an Image from Scratch

Switch to the non-root user created earlier:

```
su - devops
```

```
buildah from --name tinyc scratch
buildah containers
```

```
buildah unshare
buildah mount tinyc
```

Sample output:

```
/home/devops/.local/share/containers/storage/overlay/
2e7c3f501cc957a00e3f1cf9f2d1ae1c7f6206aa65fc0b7209628770dce4a8a4/merge
d
```

```
TINYMOUNT=`buildah mount tinyc`
```

```
echo $TINYMOUNT

ls -al $TINYMOUNT

du -sh $TINYMOUNT
```

Building an Alpine Base Image

Start by testing if you can run an Alpine-related command with the current state:

```
buildah run tinyc apk update
```

Did it work? No, it failed. Try to analyze why.

Now, download the x86_64 version of a minimal root filesystem for Alpine and extract it inside the mounted filesystem of the container created from scratch in the image above.

```
wget -c
https://dl-cdn.alpinelinux.org/alpine/v3.14/releases/x86_64/alpin
e minirootfs-3.14.0-x86_64.tar.gz

tar -xf alpine-minirootfs-3.14.0-x86_64.tar.gz -C $TINYMOUNT

ls -al $TINYMOUNT
buildah run tinyc cat /etc/issue

buildah unmount tinyc
exit

buildah config --cmd "/bin/sh" --author "xxxx yyyy" --created-by
"adding  alpine mini rootfs" tinyc
```

***NOTE**: Replace **xxxx yyyy** with your name.*

```
buildah commit tinyc alpine:3.14

buildah images
podman image history alpine:3.14

MYACCOUNT=xxxxxx
```

***NOTE**: Replace **xxxxxx** with your username/organization/project on Docker Hub to push the images.*

```
buildah tag alpine:3.14 docker.io/$MYACCOUNT/alpine:3.14

buildah login docker.io
buildah push docker.io/$MYACCOUNT/alpine:3.14
```

Building an Image with Java Runtime

Continue working with the same build container and add a Java runtime environment into it:

```
buildah from --name jre alpine:3.14
buildah run jre apk update

buildah run jre apk search java

buildah run --add-history jre apk add openjdk11-jre-headless

buildah config --author "xxxx yyyy" --created-by "installing
jre 11" jre
```

***NOTE**: Replace **xxxx yyyy** with your name.*

```
buildah commit jre docker.io/$MYACCOUNT/jre:11

podman image history docker.io/$MYACCOUNT/jre:11

podman run -it --rm docker.io/$MYACCOUNT/jre:11
```

Inside the container:

```
cat /etc/issue
java -version
exit
```

Outside the container:

```
buildah push docker.io/$MYACCOUNT/jre:11
buildah rm jre

buildah containers
buildah images
```

Test Building an Image (Step by Step)

If you are building an image with Docker, it either allows you to commit the changes that you have made to a running container, or launch a build process with a Dockerfile and run all the instructions which work towards the target stage. There is no way to take the instructions and selectively run those, while still retaining the history of changes, etc. One of the interesting features of Buildah is its ability to use instructions in Dockerfile and run them selectively and individually. You can even retain the history, commit multiple images, and do step-by-step troubleshooting. Explore all of this by using the instruction below while following along with the video lessons:

```
cd spring-petclinic
```

```
cat Dockerfile
```

Launch an intermediate build container to copy the source code to and run the compilation from:

```
buildah from --name build schoolofdevops/maven:spring
```

```
buildah containers
```

Set up a working directory (to be used by RUN, COPY, CMD) inside the build container's environment and copy over the source code to be built:

```
buildah config -h
buildah config --workingdir /app build
```

```
buildah copy -h
buildah copy build .
buildah run build ls -al /app/
```

Go ahead and compile the code with Maven:

```
buildah run build mvn package -DskipTests
buildah run build ls -al /app/target/
```

At this point, you should see a JAR file created and available inside the target directory.

The job of the build container is over (to compile the application and generate the artifact). Source code, build tools, etc., remain in this container, which you would discard later. Now, you can proceed to create the final image, which only contains Java runtime and the application artifact. You can, in fact, use the image that you created earlier.

Check presence of the Java image that you built earlier, and proceed to create a new container to package application using this image:

```
buildah images
```

```
MYACCOUNT=xxxxxx
```

***NOTE**: Replace **xxxxxx** with your username/organization/project on Docker Hub to push the images.*

```
buildah from --name package docker.io/$MYACCOUNT/jre:11
```

```
buildah containers
```

Validate that you see a new container named **package**.

Now copy the artifact generated in the build stage to the newly created **package** container:

```
buildah run build ls -al /app/target/
buildah copy -h

buildah config --workingdir /run package
buildah copy --from build package
/app/target/spring-petclinic-2.3.1.BUILD SNAPSHOT.jar

buildah run package ls -al

buildah run package mv spring-petclinic-2.3.1.BUILD-SNAPSHOT.jar

petclinic.jar buildah run package ls -al

buildah config --cmd 'java -jar petclinic.jar' --port 8080
--history-comment  'packaging petclinic app' package

buildah commit package docker.io/$MYACCOUNT/petclinic:v1

podman image history docker.io/$MYACCOUNT/petclinic:v1

podman run --name pctest --rm -idt -p 9080:8080
docker.io/$MYACCOUNT/ petclinic:v1

podman ps -l
```

Validate if the application is running on **http://IPADDRESS:9080**.

```
podman stop pctest
```

Container terminated due to the **--rm** options.

```
buildah push docker.io/$MYACCOUNT/petclinic:v1

buildah rm build package

buildah containers
buildah images
```

Skopeo

Skopeo is an interesting tool which allows you to examine a container image, and find out which files changed in which layers and how efficient your image is, etc. This can be a great starting point towards further image layers optimization. Install Skopeo and start examining the images using the following command sequence:

```
sudo apt-get -y update
sudo apt-get -y install skopeo
```

```
skopeo
skopeo inspect
docker://docker.io/postgres
```

Summary

In this lab, you learned how to use alternative tools and set up a workflow to build, run and publish your container images. This is immensely useful as the container ecosystem starts gravitating around Kubernetes, and with the requirements to support specific development workflows such as shared environments which require rootless containers, continuous integration environments with no need to mount Docker socket to build images, support for advanced image build options, etc.