

# Containers Demystified

A hands-on approach to understanding container mechanics, standards, and tooling.  
Includes Docker, Podman, and Buildah

## Class Labs

Version 2.4 by Brent Laster for Tech Skills Transformations LLC

04/09/2023

**Important Prereq:** These labs assume you have system ready with Git and Docker installed and a username/password setup on hub.docker.com (free plan is fine). If possible, run the script at <https://github.com/skillrepos/ctr-de/blob/main/image-prepare.sh> (or the pulls listed in it) to prepull images ahead of the workshop.

For any Windows systems, where you are using the Git Bash shell, you will need to prefix any Docker commands that have "-it" as options with "winpty". This implies you need to have [winpty](#) installed. Alternatively, you can run those from a standard Windows command prompt.

## Lab 1- Creating Images

**Purpose:** In this lab, we'll see how to do basic operations like building images.

1. If you haven't already, clone down the ctr-de repository from GitHub.

```
$ git clone https://github.com/skillrepos/ctr-de
```

2. Switch into the directory for our docker work.

```
$ cd ctr-de
```

3. Do an `ls` command and take a look at the files that we have in this directory.

```
$ ls
```

4. Take a moment and look at each of the files that start with "Dockerfile". See if you can understand what's happening in them.

```
$ cat Dockerfile_roar_db_image
$ cat Dockerfile_roar_web_image
```

5. Now let's build our docker database image. Type (or copy/paste) the following command: (Note that there is a space followed by a dot at the end of the command that must be there.)

```
$ docker build -f Dockerfile_roar_db_image -t roar-db .
```

6. Next build the image for the web piece. This command is similar except it takes a build argument that is the war file in the directory that contains our previously built webapp.

(Note the space and dot at the end again.)

```
$ docker build -f Dockerfile_roar_web_image --build-arg warFile=roar.war -t roar-web .
```

7. Now, let's tag our two images with your username for the docker.io repositories. We'll also give them a tag of "0.0.1" as opposed to the default tag that Docker provides of "latest".

```
$ docker tag roar-web <username>/roar-web:0.0.1  
$ docker tag roar-db <username>/roar-db:0.0.1
```

(For example, my username is bclaster, so after tagging, my images would be bclaster/roar-\*:0.0.1)

8. Do a docker images command to see the new images you've created.

```
$ docker images | grep roar
```

9. Finally, let's run our web image as a container. To do this, we'll run the container and expose a port to view it on. Execute the following command:

```
$ docker run -p 8088:8080 <username>/roar-web:0.0.1
```

10. You can open a browser session at the port we exposed and see the webapp running.

`http://localhost:8088/roar`

11. You'll notice that there is no data showing up in the table. That's because our database container is not running and being accessed. We'll see how to fix that in the next lab. In the meantime, you can stop the webapp container from running via Ctrl-C.



## END OF LAB

### Lab 2 – Composing images together

**Purpose:** In this lab, we'll see how to make multiple containers execute together with *docker compose* and use the *docker inspect* command to get information to see our running app.

1. Take a look at the docker compose file for our application and see if you can understand some of what it is doing.

```
$ cat docker-compose.yml
```

2. Run the following command to compose the two images together that we built in lab 1.

```
$ docker compose up
```

- You should see the different processes running to create the containers and start the application running. Take a look at the running containers that resulted from this command.

Note: We'll leave the *docker compose* processes running, so **open a second command prompt/terminal emulator** and enter the command below.

```
$ docker ps | grep roar
```

- Make a note of the first 3 characters of the container id (first column) for the web container (row with **roar-web** in it). You'll need those in later steps.

(For example, if the line from `docker ps` showed this:

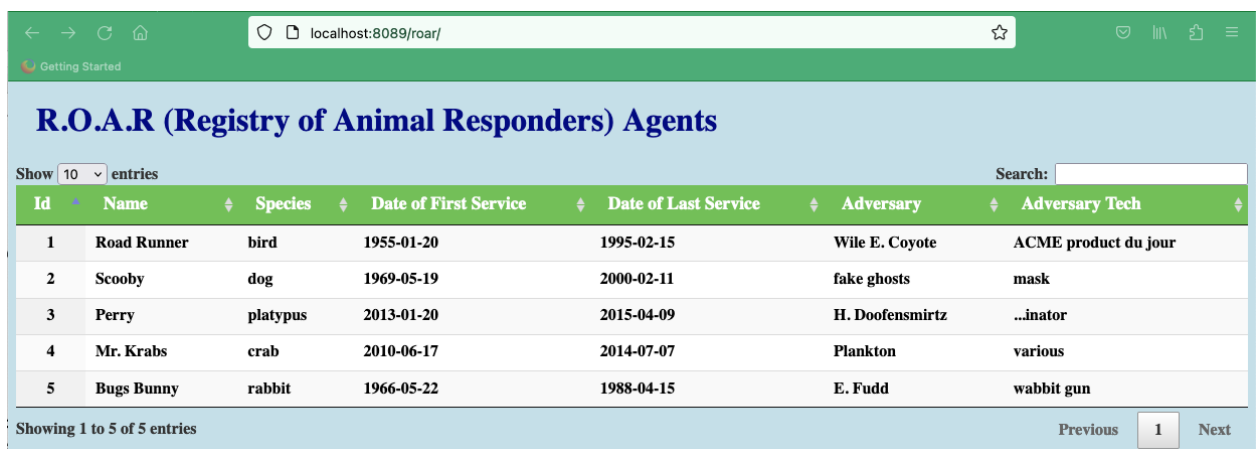
```
237a48a2aeb8    roar-web    "catalina.sh run"    About a minute ago Up About a minute 0.0.0.0:8089->8080/tcp
```

then <container id> could be "237")

- Open a web browser and go to the url below, substituting in the ip address from the step above for "<ip address>". (Note the :8080 part added to the ip address)

<http://localhost:8089/roar/>

- You should see the running app on a screen like the following:



R.O.A.R (Registry of Animal Responders) Agents						
Showing 1 to 5 of 5 entries						
Previous 1 Next						
1	Road Runner	bird	1955-01-20	1995-02-15	Wile E. Coyote	ACME product du jour
2	Scooby	dog	1969-05-19	2000-02-11	fake ghosts	mask
3	Perry	platypus	2013-01-20	2015-04-09	H. Doofensmirtz	...inator
4	Mr. Krabs	crab	2010-06-17	2014-07-07	Plankton	various
5	Bugs Bunny	rabbit	1966-05-22	1988-04-15	E. Fudd	wabbit gun

**Note:** *Some of the remaining steps call for Ctrl-C to quit running a process. If you are on Windows and using some versions of the Git Bash shell, the Ctrl-C may not work. In those cases, you can close the shell window and start a new one.*

7. Now let's find out some more details about these containers. For these commands, you can use the partial ids of either of the two containers. First, look at the top processes in this container. Then, look at the stats and note how much CPU the container is using. Ctrl-C out of that when you're done.

```
$ docker top <container id>
$ docker stats <container id>
```

8. Pause the same container's processes and look at the stats again to see the CPU usage drop off. You can also refresh the browser to see what happens when we have the container paused. Again, you can Ctrl-C when done.

```
$ docker pause <container id>
$ docker stats <container id>
```

9. When done, you can Ctrl-C again to stop the stats process and then unpause the container. If you want, you can view stats again to see the CPU usage go back to normal.

```
$ docker unpause <container id>
(optional) $ docker stats <container id>
```

### END OF LAB

## Lab 3 – Debugging Docker Containers

**Purpose:** Understanding commands and steps that we can use to learn more about what's going on in our containers if we run into problems.

1. Let's get a description of all of the attributes of our containers. For these commands, use the same 3 character container id you used in Lab 2 for the web container. (Reminder, you can use "docker ps | grep roar-web" to find the id if needed.)

Run the inspect command. Take a moment to scroll around the output.

```
$ docker inspect <container id>
```

2. Now, let's look at the logs from the running container. Scroll around again and look at the output.

```
$ docker logs <container id>
```

3. While we're at it, let's look at the history of the image (not the container).

```
$ docker history roar-web
```

4. Suppose we wanted to take a look at the actual database that is being used for the app. This is a mysql database but we don't necessarily have mysql installed. So how can we do that? Let's connect into the container and use the mysql version within the container. To do this we'll use the "docker exec" command. First find the container id of the db container.

```
$ docker ps | grep roar-db
```

5. Make a note of the first 3 characters of the container id (first column) for the db container (row with **roar-db** in it). You'll need those for the next step.
6. Now, let's exec inside the container so we can look at the actual database.

```
$ docker exec -it <container id> bash
```

Note that the last item on the command is the command we want to have running when we get inside the container – in this case the bash shell.

7. You're now "inside" the db container. Check where you are with the pwd command and then let's run the mysql command to connect to the database. (Type these at the /# prompt. Note no spaces between the options -u and -p and their arguments. You need only type the part in bold.)

```
root@container-id:/# pwd
root@container-id:/# mysql -uadmin -padmin registry
```

(Here -u and -p are the userid and password respectively and registry is the database name.)

8. You should now be at the “mysql>” prompt. Run a couple of commands to see what tables we have and what is in the database. (Just type the parts in **bold**.)

```
mysql> show tables;
mysql> select * from agents;
```

9. Exit out of mysql and then out of the container.

```
mysql> exit
root@container-id:/# exit
```

10. Since we no longer need our docker containers running or the original images around, let’s go ahead and get rid of them with the commands below.

(Hint: `docker ps | grep roar` will let you find the ids more easily)

Stop the containers

```
$ docker stop <container id for roar-web>
$ docker stop <container id for roar-db>
```

Remove the containers

```
$ docker rm <container id for roar-web>
$ docker rm <container id for roar-db>
```

Remove the images

```
$ docker rmi -f roar-web
$ docker rmi -f roar-db
```

## END OF LAB

### **Lab 4: Mapping Docker images and containers with the filesystem**

**Purpose:** In this lab, we'll explore how layers, images and containers are actually mapped and stored in the filesystem.

1. First, we need to access the underlying storage area for Docker. If you are running Docker on a Linux machine, you can open a terminal session to `"/var/lib/docker"`.

If you are on a Windows or Mac system and have Docker Desktop installed, run the following command in a terminal.

```
$ docker run -it --privileged --pid=host debian nsenter -t 1 -m -u -n -i sh
```

Now you should be able to change to the `/var/lib/docker` directory and see the files in that structure.

2. In another terminal session, let's run an interactive container based off of Ubuntu.

```
$ docker run -ti ubuntu:18.04 bash
```

3. After pulling down an instance of the image, it will be started running for you and you'll be inside the image. Let's make some simple changes so we can see how these are represented and stored in the underlying file system. We'll delete one file, create a second one and then exit the container.

```
# rm /etc/environment  
# echo new > /root/newfile.txt  
# exit
```

4. Find the first 4 characters of the ubuntu container you were working with. You can either get it from the previous steps or you can use a command like the one below to find it.

```
$ docker ps -a | grep ubuntu
```



5. Install the "jq" tool if you don't have it from <https://stedolan.github.io/jq/>

6. Run a docker inspect command to find the underlying filesystem directories for the layers - using the first 4 characters from the container id and the jq tool to get the "graphdriver" data.

```
$ docker inspect <first 4 chars of container id> | jq '.[0].GraphDriver.Data'
```

7. You should see output like the following. Take note of the value for "UpperDir". Select that and copy it.

```
{
  "LowerDir":
"/var/lib/docker/overlay2/c19f1aa7797551da6701cfe5bb716665189d7191e4bc27ba503e6eb1d3a864cc-init/diff:/var/lib/docker/overlay2/4d037a0e2bb0f50d031382246c8374382fd126b57960ff99d4b4c9be04cffd2/diff",
  "MergedDir":
"/var/lib/docker/overlay2/c19f1aa7797551da6701cfe5bb716665189d7191e4bc27ba503e6eb1d3a864cc/merged",
  "UpperDir":
"/var/lib/docker/overlay2/c19f1aa7797551da6701cfe5bb716665189d7191e4bc27ba503e6eb1d3a864cc/diff",
  "WorkDir":
"/var/lib/docker/overlay2/c19f1aa7797551da6701cfe5bb716665189d7191e4bc27ba503e6eb1d3a864cc/work"
}
```

8. In the other terminal window, where you are in the /var/lib/docker directory, do an "ls" of that directory to see what's in the Docker filesystem location.

```
$ ls <UpperDir path value copied from previous step>
```

The results should look something like this - showing the two top directories.

```
/ # ls
/var/lib/docker/overlay2/c19f1aa7797551da6701cfe5bb716665189d7191e4bc27ba503e6eb1d3a864cc/diff
    etc    root
```

9. Now, take a look at the "etc" directory and you should see the file that was removed.

```
$ ls <UpperDir path value copied from previous step>/etc
```

The results should look something like this showing the removed file.

```
/ # ls
/var/lib/docker/overlay2/c19f1aa7797551da6701cfe5bb716665189d7191e4bc2
7ba503e6eb1d3a864cc/diff/etc
environment
```

10. Next, look at the "root" directory and you should see the file that was created.

```
$ ls <UpperDir path value copied from previous step>/root
```

The results should look something like this showing the added file.

```
/ # ls
/var/lib/docker/overlay2/c19f1aa7797551da6701cfe5bb716665189d7191e4bc2
7ba503e6eb1d3a864cc/diff/root
newfile.txt
```

11. If you want to see where the original image is stored, grab the second path under the "LowerDir" section (after the "init/diff:" piece). It is highlighted below.

```
{
  "LowerDir":
    "/var/lib/docker/overlay2/c19f1aa7797551da6701cfe5bb716665189d7191e4bc
    27ba503e6eb1d3a864cc-
    init/diff:/var/lib/docker/overlay2/4d037a0e2bb0f50d031382246c8374382fd
    d126b57960ff99d4b4c9be04cffd2/diff",
  "MergedDir":
    "/var/lib/docker/overlay2/c19f1aa7797551da6701cfe5bb716665189d7191e4bc
    27ba503e6eb1d3a864cc/merged",
  "UpperDir":
    "/var/lib/docker/overlay2/c19f1aa7797551da6701cfe5bb716665189d7191e4bc
    27ba503e6eb1d3a864cc/diff",
  "WorkDir":
    "/var/lib/docker/overlay2/c19f1aa7797551da6701cfe5bb716665189d7191e4bc
    27ba503e6eb1d3a864cc/work"
}
```

12. You can do an "ls" on the path copied from the previous step and you'll see the original starting layer for the container. You can also look at the

"etc" and "root" directories to see the original state of those without the changes we made. Afterwards, you can stop the container that was started in step 1.

```
$ ls <path value from 2nd part of LowerDir copied from previous step>
```

```
$ ls <path value from 2nd part of LowerDir copied from previous step>/root
```

```
$ ls <path value from 2nd part of LowerDir copied from previous step>/etc
```

### END OF LAB

**Note for labs 5 & 6:** If you are running on Windows, in the Git Bash shell, you will likely need to use a specific syntax for the -v specification in the docker run. For the "<working dir>" piece, you would use "\$(\$pwd)" to represent the current directory, as in

```
winpty docker run -it --device /dev/fuse:rw --privileged -v /$(pwd)/ctr-de:/build -p 8087:8080 quay.io/buildah/stable bash
```

If you are running on Windows, and using a standard Windows command prompt, use the full absolute path to the directory, as in

```
C:\Users\brent>docker run -it --device /dev/fuse:rw --privileged -v c:\users\brent\ctr-de:/build quay.io/buildah/stable bash
```

## **Lab 5 - Working with Podman**

**Purpose:** In this lab, we'll get a chance to work with Podman, an alternative to Docker that also includes the abilities to group and work with containers in "pods".

1. We need an instance of the podman application to work with. The easiest way to do this is to run it via a container via the instructions below (<working dir> refers to the path on your system where you are working (where you cloned the repo to) - for example ~/ctr-de:

```
$ docker run -it --device /dev/fuse:rw --privileged -v <working dir>:/work quay.io/podman/stable bash
```

2. Check that podman is installed and responding.

```
$ podman version
```

3. Now that you have podman installed, clone down the repository for us to work with in building images and then change into the directory with the docker content.

```
$ cd work
```

4. Now, build the two images (the web one and the database one) that we need for our application. Note that the syntax for podman is just like the syntax for Docker. Afterwards, you can see the images with podman.

```
$ podman build -t roar-web:1.0.0 --build-arg warFile=roar.war -f Dockerfile_roar_web_image .
```

```
$ podman build -t roar-db:1.0.0 -f Dockerfile_roar_db_image .
```

```
$ podman images
```

5. Now let's create a pod.

```
$ podman pod create --name roar-pod -p 8087:8080 --network bridge
```

6. Next, we'll list the pod we have and then inspect it to look at it closer.

```
$ podman pod ls
```

```
$ podman inspect roar-pod
```

7. Notice the inspect lists one container at the bottom. Let's look closer at what that container is.

```
$ podman ps -a --pod
```

8. Add the web image as a container to the pod.

```
$ podman run --pod roar-pod --name roar-web --ipc=private -d roar-web:1.0.0
```

9. Finally, we'll add the database image as a container to the pod.

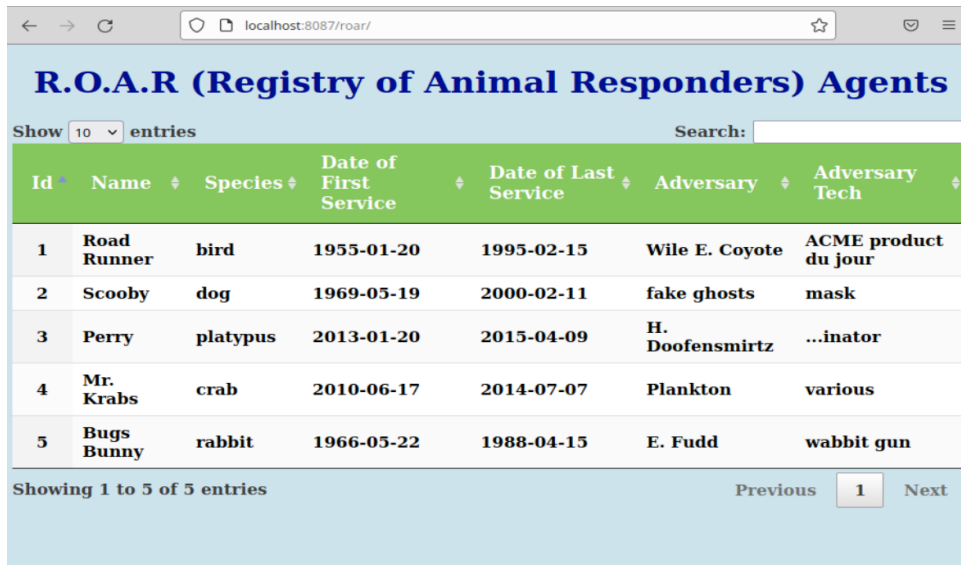
```
$ podman run --pod roar-pod --name roar-db --env-file env.list --ipc=private -d
roar-db:1.0.0
```

10. You can now see the containers running in the pod.

```
$ podman inspect roar-pod
```

11. (optional) Now you can open up the url below in a browser and see the application running.

<http://localhost:8087/roar>



R.O.A.R (Registry of Animal Responders) Agents						
Show 10 entries		Search:				
Id	Name	Species	Date of First Service	Date of Last Service	Adversary	Adversary Tech
1	Road Runner	bird	1955-01-20	1995-02-15	Wile E. Coyote	ACME product du jour
2	Scooby	dog	1969-05-19	2000-02-11	fake ghosts	mask
3	Perry	platypus	2013-01-20	2015-04-09	H. Doofensmirtz	...inator
4	Mr. Krabs	crab	2010-06-17	2014-07-07	Plankton	various
5	Bugs Bunny	rabbit	1966-05-22	1988-04-15	E. Fudd	wabbit gun

Showing 1 to 5 of 5 entries

Previous 1 Next

12. Leave podman running for the next lab!

END OF LAB

**Note for lab 6:** If you hit an error like this "ERRO[0000] Unmounting..." in step 5, then do this command in the buildah terminal and then rerun.

```
# export STORAGE_OPTS="overlay.mount_program=/usr/bin/fuse-overlayfs"
```

## Lab 6: Working with Buildah

**Purpose:** In this lab, we'll work with the container build and management tool Buildah.

1. With podman still running, open/go to a separate terminal.
2. We need an instance of the Buildah application to work with. The easiest way to do this is to run it via a container via the instruction below ( **<working dir>** refers to the path on your system where you are working (where you cloned the repo to) - for example **~/ctr-de**:

```
$ docker run -it --device /dev/fuse:rw --privileged -v <working dir>:/build quay.io/buildah/stable bash
```

3. You'll now be in the container with access to buildah. Go to the build directory that you mounted into the container.

```
$ cd /build
```

4. In this directory, we have a shell script that will run buildah commands to produce images (instead of using Docker and a Dockerfile). Take a look at the script. Notice that it is using buildah to run each individual command that we would normally have in a Dockerfile.

```
$ cat buildah-roar.sh
```

5. Now, run the script. We will tell it to pull images from Docker and populate our database one with test data. We also need to pass in the built deliverable to be pulled in for the webapp. That's what **roar.web** is.

```
$ bash ./buildah-roar.sh docker test roar.war
```

6. After the images are created, you'll see them listed. But you can also see the entire list of current images via the "buildah images" command.

```
$ buildah images
```

7. For this next step, you will need your docker userid. Login using the buildah login command and your username/password.

```
$ buildah login docker.io
```

8. After logging in to the registry, tag your images replacing "localhost" with your username.

```
$ buildah tag localhost/roar-web:1.0.1 <username>/roar-web:1.0.1  
$ buildah tag localhost/roar-db:1.0.1 <username>/roar-db:1.0.1
```

(For example my images on docker would be bclaster/roar-db:1.0.1 since my username on docker is bclaster)

9. Push the images out to the registry - substituting your actual username for <username>.

```
$ buildah push <username>/roar-web:1.0.1  
$ buildah push <username>/roar-db:1.0.1
```

10. Now you can switch back to the terminal where podman is running.

11. Use podman to pull the database image (roar-db) that you just pushed. (Note that you may be asked to select the container registry image - for example, for docker, it would be docker.io/<username>/roar-db:1.0.1)

```
$ podman pull <username>/roar-db:1.0.1
```

12. Take a look at the containers you have locally and note the ids of the roar-db one you just pulled and note the id of the previous 1.0.0 version of the roar-db one as we're going to replace it.

```
$ podman container list --all | grep roar-db:1.0.0
```

(You want the first 4 characters of the CONTAINER ID from the first column.)

13. Now, we'll use podman to remove the old container from the pod and replace it with the new one.

```
$ podman container stop <first 4 chars of container id for roar-db:1.0.0>
```

