Exercise: Container Registry (BONUS)

Docker has a very useful feature that uses this layered union-mounted filesystem.

When we build another container image whose Dockerfile has lines in common with a Dockerfile we've already built against, Docker keeps track of what filesystem layer contained the changes made by each step in the Dockerfile, and skips running the command when it knows what the results would be. We'll explore this here.

NOTE: when an instruction shows you the results from our test system, it often won't match your machine exactly down to the numbers, especially when time units like seconds are involved. Don't worry about this.

1. Start up a fresh lxterminal by clicking the "sparrow" logo in the bottom-left corner of the screen, clicking run, typing `lxterminal` and hitting enter. Alternatively, use the hot key sequence below:

```
<hold down Alt><hit F2>lxterminal<HIT the enter key>
```

2. Log in to the docker virtual machine with password `logidebtech`:

```
ssh user@docker
logidebtech
```

3. Install an NTP daemon on this host. Then make sure you have the `gcr.io/distroless/base` image on this host - we have chosen it for its small filesystem size. `docker pull` retrieves the image from the `gcr.io` image registry and caches it on this machine's Docker cache.

```
sudo apt -y install ntp
logidebtech
docker pull gcr.io/distroless/base
```

4. Start the Docker registry now on this host.

```
docker run --name=registry -d  -p 5000:5000 -e REGISTRY_STORAGE_FILESYSTEM_ROOTDIRECTORY=/da
```

5. Tag the `distroless/base` image you already have as `distroless-base`, but on your private local registry, where it will thus be named `localhost:5000/distroless-base`:

```
docker tag gcr.io/distroless/base localhost:5000/distroless-base
```

6. Push this image up to your local registry:

```
docker push localhost:5000/distroless-base
```

7. Now delete your local copy of Docker Hub's `gcr.io/distroless/base`, but look at the free disk space before and after:

```
df -m /
docker rmi gcr.io/distroless/base
df -m /
```

8. Notice that the operation didn't really free up space – here's the output from our test system:

```
user@cfs:~$ df -m /
Filesystem     1M-blocks  Used Available Use% Mounted on
/dev/vda1           5949  2265      3408  40% /
user@cfs:~$ docker rmi gcr.io/distroless/base
Untagged: gcr.io/distroless/base:latest
Untagged: gcr.io/distroless/base@sha256:ce8bc342dd7eeb0baccbef2ce00afc0c72af1ea166794f55ef8f
user@cfs:~$ df -m /
Filesystem     1M-blocks  Used Available Use% Mounted on
/dev/vda1           5949  2265      3408  40% /
```

9. Now delete your local copy of your registry's `distroless-base` image using the `docker rmi` (remove image) command. Check the free disk space before and after:

```
df -m /
docker rmi localhost:5000/distroless-base
df -m /
```

10. Notice that this image deletion did free up space and that this image remove command had two extra lines of output, saying that layers were deleted. Here's output from our test system:

```
user@cfs:~$ df -m /
Filesystem     1M-blocks  Used Available Use% Mounted on
/dev/vda1           5949  2265      3408  40% /
user@cfs:~$ docker rmi localhost:5000/distroless-base
Untagged: localhost:5000/distroless-base:latest
Untagged: localhost:5000/distroless-base@sha256:9ad1a806eaf6eaf7ce3dc5dc4bdd8bc2d215e6182e45
Deleted: sha256:2e9fdb5bbbcb2dfd7807b928a2fabd5df5cfe3f02983c9b7e2c6263f0a9eedd1
Deleted: sha256:28bf1ce335eadb04b11046842219e9a703203ad454126c491140a68f7ffe93d8
Deleted: sha256:0b031aac65698c8794dc6bc317a45589e07bc2db1421178f30a2c7f69a4a2cf5
user@cfs:~$ df -m /
Filesystem     1M-blocks  Used Available Use% Mounted on
/dev/vda1           5949  2241      3432  40% /
```

11. Think about what's happening here. Docker is saving space by keeping a hash of each image layer and simply tagging one or more layers with

whatever name we tag it with. So the `distroless/base` image layers aren't deleted until no tags refer to them.

12. Now, let's pull this image back down from our local registry:

```
docker pull localhost:5000/distroless-base
```

13. Let's create an image based on this one. Switch directory to `/home/user/imagedev/`:

```
cd /home/user/imagedev/
```

14. Create a `Dockerfile` in this directory.

```
cat <<EOF >Dockerfile
FROM localhost:5000/distroless-base
COPY Dockerfile /usr/share
EOF
```

15. This Dockerfile says we'll start with the `distroless/base` image you just pushed to the repository, then copies the current directory's Dockerfile into it.

```
FROM localhost:5000/distroless-base
COPY Dockerfile /usr/share
```

16. Build a container from this image with the `docker build` command, which takes a name (a tag) and a directory in which to find a `Dockerfile` file. We'll call this image `localhost:5000/base-plus-dockerfile`.

```
docker build -t localhost:5000/base-plus-dockerfile .
```

17. Now let's build a more useful container image. Change directory to the `build-with-du` subdirectory:

```
cd /home/user/imagedev/build-with-du
```

18. Display the `Dockerfile` in this directory.

```
cat Dockerfile
```

19. Note how this Dockerfile starts with the `localhost:5000/distroless-base` image, then copies a `du` binary into `/bin`. It also uses two more directives, `ENTRYPOINT` and `CMD`, to specify a program to run when the container starts, along with any arguments passed in on the command line. Here's the sample output on our system:

```
FROM localhost:5000/distroless-base
COPY du bin/
ENTRYPOINT ["/bin/du","-ks"]
CMD ["/bin"]
```

20. Build a container from this image with the `docker build` command, which takes a tag and a directory in which to find a `Dockerfile` file.

```
docker build -t localhost:5000/base-plus-du ./
```

21. Let's start a container based on `localhost:5000/base-plus-du`, using the `-d` (detach) flag to detach from the container's stdio. We'll name the container `ctr`:

```
docker run -d --name=ctr localhost:5000/base-plus-du
```

22. This container has completed and exited, but we can go look at its output via the `docker logs` command:

```
docker logs ctr
```

23. The output you see will be the size of the `/bin/` directory in bytes. Delete the container now:

```
docker rm ctr
```

24. Note that the container's output gave us the disk usage of the `/bin` directory in kilobytes. What if we wanted this image to do the same thing, using `/bin` as a default directory to measure, but allowing the user to specify a different directory, say `/usr`, without having to rebuild the image? This is exactly what `CMD` does in the Dockerfile. It indicates arguments that you can override easily. by putting them on the end of the `docker run` command line. Let's remove the `-d` flag, so we can see the output in real time and add a `-rm` flag, so the container is destroyed as soon as it exits:

```
docker run --rm --name=ctr2 localhost:5000/base-plus-du /usr
```

25. Note that we now see the total size of the `/usr` directory in kilobytes. So, you've seen how `ENTRYPOINT` and `CMD` interplay. Summarizing:

- `ENTRYPOINT` tells Docker what program to run when this container starts. It can optionally include arguments. These arguments aren't easily overriden, unless the entire entrypoint program is replaced.
- `CMD` indicates arguments that Docker should add to the command line created from `ENTRYPOINT`. These arguments are intended to be easily overwritten, the way we overwrote `/bin` with `/usr`.
- This produces the situation where the command run as the container's first process will start with `du -ks` and end with either `/bin` or whatever is placed after the image name on the `docker run` command line.

26. Let's push this container image we've built to our local registry - the output should be interesting:

```
docker push localhost:5000/base-plus-du
```

27. Note that the output shows that Docker didn't have to push the two of the layers to the registry, as the registry already had them! Our sample output follows:

```
Using default tag: latest
The push refers to repository [localhost:5000/base-plus-du]
7d221ee8ae69: Pushed
f89ce21aca6a: Mounted from distroless-base
0b031aac6569: Mounted from distroless-base
latest: digest: sha256:52ee4f9b7565d65f3c2db68afd97384ebadbe0899f0f6076ce2c5c43489550b6 size
```

28. That will certainly make things faster, especially when we're pushing to an Internet-connected registry like Docker Hub! Let's delete the `base-plus-du` image from our local image cache - we'll leave it up on the registry, of course:

```
docker rmi localhost:5000/base-plus-du
```

29. Now, let's pull down the container image again.

```
docker pull localhost:5000/base-plus-du
```

30. Note that Docker didn't have to pull down some of the layers - the ones that were part of the `distroless-base` that it still had cached. Here's the sample output from our machine:

```
Using default tag: latest
latest: Pulling from base-plus-du
36698cfa5275: Already exists
6a8659ec8836: Already exists
7cf3941d8a27: Already exists
Digest: sha256:52ee4f9b7565d65f3c2db68afd97384ebadbe0899f0f6076ce2c5c43489550b6
Status: Downloaded newer image for localhost:5000/base-plus-du:latest
localhost:5000/base-plus-du:latest
```

31. Imagine that a colleague of yours had already cached `distroless-base` and wanted to download `base-plus-du` the way you just did. Their download time would be greatly reduced because they only need to pull down this one layer of the image: the layer that represents these lines from the Dockerfile:

```
COPY du bin/
ENTRYPOINT ["/bin/du","-ks"]
CMD ["/bin"]
```

32. Let's run a container to see one more Docker feature: volume mounting. Start a new container based on `base-plus-du`, but to mount the host's `/usr` onto the container's `/mnt` directory using `-v`, and tell the container to do a disk usage tally of that directory:

```
docker run -v /usr:/mnt --rm --name=ctr3 localhost:5000/base-plus-du /mnt
```

33. Note that this operation took a little longer to run. It also showed the enormous size difference of your the virtual machine's `/usr` directory, versus that of the container's `/usr` directory. Here's the output of this command:

```
1476708 /mnt
```

34. Finally, let's build the container image from the `/home/user/imagedev/busybox-from-scratch` directory, then explore it with an interactive shell. Here are three commands to build and run the image:

```
cd /home/user/imagedev/busybox-from-scratch
docker build -t localhost:5000/busyboxfromscratch .
```

35. Stop for a second to check how much smaller the last couple images you've built are than `centos` was:

```
docker images | egrep '(centos|busybox|base-plus-du)'
```

36. Notice that the two images you built are about one tenth (0.1) times the size of the `centos:7` image. Here's the sample output from our system:

```
localhost:5000/busyboxfromscratch     latest    4cbcffbfe310   2 minutes ago    21.2MB
localhost:5000/base-plus-du           latest    750eabf6a6b2   31 minutes ago   20.4MB
centos                                7         eeb6ee3f44bd   6 months ago     204MB
```

37. Now explore the busybox image via an interactive shell in a running container - feel free to run a few commands in the container, but don't exit the shell:

```
docker run -it --name=busybox localhost:5000/busyboxfromscratch /bin/sh
```

38. Detach from the container's shell by hitting CTRL-P-Q.

39. Run a `docker inspect` command to see how you might get the details for a container, like it's IP address:

```
docker inspect busybox
```

40. Notice that the end of the output shows the container's IP address. After class, you can parse the rest of the output with `jq` if you'd like. You don't need to - just realize this is part of how Kubernetes will be orchestrating, say, networking, for thousands of containers. Here's the end of our sample output on that last command:

```
                "IPAddress": "172.17.0.3",
                "IPPrefixLen": 16,
                "IPv6Gateway": "",
                "GlobalIPv6Address": "",
                "GlobalIPv6PrefixLen": 0,
                "MacAddress": "02:42:ac:11:00:03",
                "DriverOpts": null
            }
        }
    }
]
```