Docker Fundamentals Exercises

Contents

1 Running & Inspecting a Container 4

1.1 Running Containers 4

1.2 Listing Containers 5

1.3 Conclusion 5

2 Interactive Containers 6

2.1 Writing to Containers 6

2.2 Reconnecting to Containers 6

2.3 More Container Listing Options 7

2.4 Conclusion 7

3 Detached Containers and Logging 7

3.1 Running a Container in the Background 7

3.2 Attaching to Container Output 8

3.3 Logging Options 8

3.4 Conclusion 8

4 Starting, Stopping, Inspecting and Deleting Containers 9

4.1 Starting and Restarting Containers 9

4.2 Inspecting a Container with `docker container inspect` 9

4.3 Deleting Containers 10

4.4 Conclusion 10

5 Interactive Image Creation 10

5.1 Modifying a Container 10

5.2 Capturing Container State as an Image with `docker container commit` 11

5.3 Conclusion 11

6 Creating Images with Dockerfiles (1/2) 12

6.1 Writing and Building a Dockerfile 12

6.2 The Build Cache 12

6.3 The `history` Command 12

6.4 Conclusion 13

7 Creating Images with Dockerfiles (2/2) 13
7.1 Default Commands via `CMD` 13
7.2 Default Commands via `ENTRYPOINT` 13
7.3 `CMD` and `ENTRYPOINT` Together 14
7.4 Conclusion 14

8 Dockerizing an Application 14
8.1 Setting up a Java App 14
8.2 Dockerize your App 15
8.3 Restructure your Application 15
8.4 Conclusion 16
8.5 Optional Exercise 17

9 Managing Images 17
9.1 Tagging and Listing Images 17
9.2 Sharing Images on Dockerhub 17
9.3 Private Images 18
9.4 Conclusion 18

10 Cleanup Commands 19

11 Inspection Commands 19
11.1 System Information 19
11.2 System Events 20
11.3 Summary 21

12 Image Autobuilds 21
12.1 Setting up a Code Repo 21
12.2 Starting Automated Builds 22
12.3 Automated Build in Action 22
12.4 Conclusion 23

13 Creating and Mounting Volumes (1/3) 23
13.1 Creating a Volume 23
13.2 Volumes During Image Creation 23
13.3 Conclusion 24

14 Creating and Mounting Volumes (2/3) 24
14.1 Finding the Host Mountpoint 24
14.2 Conclusion 25

15 Creating and Mounting Volumes (3/3) 25
15.1 Deleting Volumes 25
15.2 Conclusion 25

16 Mounting Host Directories 25
16.1 Conclusion 26

17 Volumes Usecase: Recording Logs 26
17.1 Set up an app with logging 26
17.2 Inspect Logs on the Host 27
17.3 Conclusion 27

18 Sharing Volumes 27
18.1 Conclusion 28

19 Docker Plugins 28
19.1 Installing a Plugin 28
19.2 Enabling and Disabling a Plugin 29
19.3 Inspecting a Plugin 29
19.4 Removing a Plugin 30
19.5 Using the Plugin 30
19.6 Summary 30

20 Introduction to Container Networking 31
20.1 The Default Bridge Network 31
20.2 User Defined Bridge Networks 32
20.3 Inter-Container Communication 32
20.4 Conclusion 33

21 Container Port Mapping 33
21.1 Port Mapping at Runtime 33
21.2 Exposing Ports from the Dockefile 34
21.3 Conclusion 34

22 Starting a Compose App 34
22.1 Prepare Service Images 35
22.2 Start the App 35
22.3 Viewing Logs 36
22.4 Conclusion 36

23 Scaling a Compose App 36
23.1 Scaling a Service 36
23.2 Scale Nonlinearity 37
23.3 Conclusion 37

24 Creating a Swarm 37
24.1 Start Swarm Mode 37
24.2 Add Workers to the Swarm 38
24.3 Promoting Workers to Managers 38
24.4 Conclusion 38

25 Starting a Service 39
25.1 Start a Service 39
25.2 Scaling a Service 39

25.3 Cleanup 39
25.4 Conclusion 40

26 Load Balancing & the Routing Mesh 40
26.1 Deploy a service 40
26.2 Observe load-balancing and Scale 40
26.3 The Routing Mesh 41
26.4 Cleanup 41
26.5 Conclusion 41

27 Node Failure Recovery 42
27.1 Set up a Service 42
27.2 Simulate Node Failure 42
27.3 Cleanup 42
27.4 Conclusion 42

28 Dockercoins On Swarm 42
28.1 Prepare Service Images 43
28.2 Start our Services 43

29 Scaling and Scheduling Services 43
29.1 Scaling up a Service 43
29.2 Scheduling Services 44
29.3 Conclusion 45

30 Updating a Service 45
30.1 Rolling Updates 45
30.2 Parallel Updates 45
30.3 Shutting Down a Stack 46

31 Docker Secrets 46
31.1 Creating Secrets 46
31.2 Creating secrets with labels 47
31.3 Listing and inspecting secrets 47
31.4 Deleting Secrets 48
31.5 Using Secrets 48
31.6 Updating a Secret 49
31.7 Preparing an image for use of secrets 49
31.8 Conclusion 50

1 Running & Inspecting a Container

1.1 Running Containers

First, let's start a container, and observe the output:

```
docker container run ubuntu:14.04 echo "hello world"
```

The `ubuntu:14.04` part indicates the *image* we want to use to define this container; it includes the underlying operating system and its entire filesystem. `echo "hello world"` is the command we want to execute inside the context of the container defined by that image. Once the image downloads, the container is created and the specified command is executed within that container.

Now create another container from the same image, and run a different command inside of it:

```
docker container run ubuntu:14.04 ps -ef
```

Notice the much faster execution time compared to the first container that was run. This is due to the fact that Docker now has the Ubuntu 14.04 image locally and thus does not need to download the image again.

Also, notice that `ps -ef` was PID 1 inside the container; try doing `ps -ef` at the host prompt and see what process is PID 1. A container lives in a namespaced portion of the host's kernel, with its own process tree, user ID spectrum, network stack and more; PID 1 in a container is always the command at the end of `docker container run` - `ps -ef` in the example above.

1.2 Listing Containers

Try listing all your currently running containers:

```
docker container ls
```

There's nothing listed, since the containers you ran executed a single command, and shut down when finished. This is by desgin; containers are meant to be *ephemeral*, created to perform a single task, stopped when that task is complete, and re-created in future if that task needs to run again. More specifically: when a container's PID 1 process exits, the container stops. List stopped as well as running containers with the `-a` flag:

```
docker container ls -a
```

1.3 Conclusion

In this exercise you ran your first container using `docker container run`, and explored the importance of the PID 1 process in a container. You also saw how to list containers on your Docker host machine, using `docker container ls`.

2 Interactive Containers

2.1 Writing to Containers

Create a container using the ubuntu:14.04 image, and connect to its bash shell in interactive mode using the -i flag (also the -t flag, to indicate a tag name):

```
docker container run -it ubuntu:14.04 bash
```

Explore your container's filesystem with ls, and then create a new file:

```
ls
touch test.dat
ls
```

Exit the container:

```
exit
```

Run the same command as above to start a container in the same way:

```
docker container run -it ubuntu:14.04 bash
```

Try finding your test.dat file inside this new container; it is nowhere to be found. Exit this container for now:

```
exit
```

2.2 Reconnecting to Containers

We'd like to recover the information written to our container in the first example, but starting a new continer didn't get us there; instead, we need to restart our original container, and reconnect to it. List all your stopped containers:

```
docker container ls -a
```

We can restart a container via the Container ID listed in the first column. Use the container ID for the first ubuntu:14.04 container you created with bash as its command:

```
docker container start 20529fc47a36
```

Reconnect to your container with docker container exec (this can be used to execute *any* command in a running Docker container):

```
docker container exec -it 20529fc47a36 bash
```

List the contents of the container's filesystem again with ls; your test.dat should be where you left it. Exit the container again by typing exit.

2.3 More Container Listing Options

In the last step, we saw how to get the short container ID of all our containers using `docker container ls -a`. Try adding the `--no-trunc` flag to see the entire container ID:

```
docker container ls -a --no-trunc
```

This long ID is the same as the string that is returned after starting a container with `docker container run`.

A couple of other flags you may want to know off-hand (try each to see their output):

- `-q`, for listing only the container ID (notice flags can be combined):

```
docker container ls -aq
```

- `-l`, for listing the last container to have started:

```
docker container ls -l
```

Finally, you can also filter results with the `--filter` flag; for example, try filtering by exit code:

```
docker container ls -a --filter "exited=1"
docker container ls -a --filter "exited=0"
```

2.4 Conclusion

In this demo, you saw that files added to a container's filesystem do not get added to all containers created from the same image; changes to a container's filesystem are local to itself. You also learned how to restart a stopped Docker container using `docker container start`, how to run a command in a running container using `docker container exec`, and also saw some more options for listing containers via `docker container ls`.

3 Detached Containers and Logging

3.1 Running a Container in the Background

First try running a container as usual; the STDOUT and STDERR streams from whatever is PID 1 inside the container is directed to the terminal:

```
docker container run ubuntu:14.04 ping 127.0.0.1 -c 10
```

The same process can be run in the background with the `-d` flag:

```
docker container run -d ubuntu:14.04 ping 127.0.0.1
```

Find this second container's ID, and use it to inspect the logs it generated:

```
docker container logs <container ID>
```

These logs correspond to STDOUT and STDERR from the container's PID 1.

3.2 Attaching to Container Output

We can attach a terminal to a container's PID 1 output with the `attach` command; try it with the last container you made in the previous step:

```
docker container attach <container ID>
```

We can leave attached mode by then pressing `CTRL+C`. After doing so, list your running containers; you should see that the container you attached to has been killed, since the `CTRL+C` issued killed PID 1 in the container, and therefore the container itself.

Try running the same thing in detached interactive mode:

```
docker container run -d -it ubuntu:14.04 ping 127.0.0.1
```

Now attach to this container like you did the first one, but this time detach with `CTRL+P+Q`, and list your running containers. In this case, the container should still be happily running in the background after detaching from it.

3.3 Logging Options

We saw previously how to read the entire log of a container's PID 1; we can also use a couple of flags to control what logs are displayed. `--tail n` limits the display to the last `n` lines; try it with the container that should be running from the last step:

```
docker container logs --tail 5 <container ID>
```

We can also follow the logs as they are generated with `-f`:

```
docker container logs -f <containerID>
```

(`CTRL+C` to break out of following mode). Finally, try combining the tail and follow flags to begin following the logs from a point further back in history.

3.4 Conclusion

In this scenario, we saw how to run processes in the background, attach to them, and inspect their logs. We also saw an explicit example of how killing PID 1 in a container kills the container itself.

4 Starting, Stopping, Inspecting and Deleting Containers

4.1 Starting and Restarting Containers

Start by running a tomcat server in the background, and check that it's really running:

```
docker container run -d tomcat
docker container ls
```

Stop the container using `docker container stop`, and check that the container is indeed stopped:

```
docker container stop <container ID>
docker container ls -a
```

Start the container again with `docker container start`, and attach to it at the same time:

```
docker container start -a <containerID>
```

Detach and stop the container with `CTRL+C`, then restart the container without attaching and follow the logs starting from 10 lines previous. Finally, stop the container with `docker container kill`:

```
docker container kill <containerID>
```

Both `stop` and `kill` send a SIGKILL to PID 1 in the container; the difference is that `stop` first sends a SIGTERM, then waits for a grace period (default 10 seconds) before sending the SIGKILL, while `kill` fires the SIGKILL immediately.

4.2 Inspecting a Container with `docker container inspect`

Start your tomcat server again, then inspect the container details using `docker container inspect`:

```
docker container inspect <container ID>
```

Find the container's IP and long ID in the JSON output of `inspect`. If you know the key name of the property you're looking for, try piping to grep:

```
docker container inspect <container ID> | grep IPAddress
```

Now try grepping for `Cmd`, the PID 1 command being run by this container. `grep`'s simple text search doesn't always return helpful results.

Another way to filter this JSON is with the `--format` flag. Syntax follows Go's text/template package: http://golang.org/pkg/text/template/. For example, to find the `Cmd` value we tried to grep for above, instead try:

```
docker container inspect --format='{{.Config.Cmd}}' <container ID>
```

Keys nested in the JSON returned by `docker container inspect` can be chained together in this fashion. Try modifying this example to return the IP address you grepped for previously.

Finally, we can extract all the key/value pairs for a given object using the `json` function:

```
docker container inspect --format='{{json .Config}}' <container ID>
```

4.3 Deleting Containers

Start three containers in background mode, then stop the first one.

List only exited containers using the `--filter` flag we learned earlier, and the option `status=exited`.

Delete the container you stopped above with `docker container rm`, and do the same listing operation as above to confirm that it has been removed:

```
docker container rm <container ID>
docker container ls ...
```

Now do the same to one of the containers that's still running; notice `docker container rm` won't delete a container that's still running, unless we pass it the force flag `-f`. Delete the second container you started above:

```
docker container rm -f <container ID>
```

Try using the `docker container ls` flags we learned previously to remove the last container that was run, or all stopped containers. Recall that you can pass the output of one shell command `cmd-A` into a variable of another command `cmd-B` with syntax like `cmd-B $(cmd-A)`.

4.4 Conclusion

In this scenario, you learned how to use `docker container start`, `stop`, `rm` and `kill` to start, stop and delete containers. You also saw the `docker container inspect` command, which returns metadata about a given container.

5 Interactive Image Creation

5.1 Modifying a Container

Start a bash terminal in an ubuntu container:

```
docker container run -it ubuntu:14.04 bash
```

Install a couple pieces of software in this container - there's nothing special about `vim` and `wget`, any changes to the filesystem will do. Afterwards, exit the container:

```
apt-get update
apt-get install -y wget vim
exit
```

Finally, try `docker container diff` to see what's changed about a container relative to its image; you'll need to get the container ID via `docker container ls -a` first:

```
docker container ls -a
docker container diff <container id>
```

Make sure the results of the diff make sense to you before moving on.

5.2 Capturing Container State as an Image with `docker container commit`

Installing wget and vim in the last step wrote information to the container's read/write layer; now let's save that read/write layer as a new read-only image layer in order to create a new image that reflects our additions, via the `docker container commit`:

```
docker container commit <container id> <your dockerhub ID>/myapp:1.0
```

Check that you can see your new image by listing all your images:

```
docker image ls
```

Create a container running bash using your new image, and check that vim and wget are installed:

```
docker container run -it <your dockerhub ID>/myapp:1.0 bash
which vim
which wget
```

Create a file in your container and commit that as a new image. Use the same image name but tag it as 1.1.

Finally, run `docker container diff` on your most recent container; does the output make sense? What do you guess the prefixes A, C and D at the start of each line mean?

5.3 Conclusion

In this exercise, you saw how to inspect the contents of a container's read / write later with `docker container diff`, and commit those changes to a new image layer with `docker container commit`.

6 Creating Images with Dockerfiles (1/2)

6.1 Writing and Building a Dockerfile

Create a folder called `myimage`, and a text file called `Dockerfile` within that folder.

In `Dockerfile`, include the following instructions:

```
FROM ubuntu:16.04

RUN apt-get update
RUN apt-get install -y iputils-ping
```

Build your image with the `build` command:

```
docker image build -t <username>/myimage .
```

Verify that your new image exists with `docker image ls`, then use it to run a container and ping something from within that container.

6.2 The Build Cache

In order to speed up image builds, Docker preserves a cache of previous build steps. Try running the exact same build command you did previously:

```
docker image build -t <username>/myimage .
```

Each step should execute immediately and report `using cache` to indicate that the step was not repeated, but fetched from the build cache.

Now open your Dockerfile and add another `RUN` step at the end to install `vim`.

Build the image again as above; which steps is the cache used for?

Build the image again; which steps use the cache this time?

Finally, swap the order of the two `RUN` commands for installing `ping` and `vim` in the Dockerfile, and build one last time. Which steps are cached this time?

6.3 The `history` Command

The `docker image history` command allows us to inspect the build cache history of an image. Try it with your new image:

```
docker image history <image id>
```

Note the image id of the layer built for the `apt-get update` command. Now replace the two `RUN` commands that installed `iputils-ping` and `vim` with a single command:

```
RUN apt-get install -y iputils-ping vim
```

Build the image again, and run `docker image history` on this new image. How has the history changed?

6.4 Conclusion

So far, we've seen how to write a basic Dockerfile using FROM and RUN commands, some basics of how image chaching works, and seen the docker image history command. After some discussion, we'll see how to define some defualt commands and options for running as the PID 1 of containers created from an image defined by our Dockerfile.

7 Creating Images with Dockerfiles (2/2)

7.1 Default Commands via CMD

Add the following line to your Dockerfile from the last problem, at the bottom:

```
CMD ["ping", "127.0.0.1", "-c", "30"]
```

This sets ping as the default command to run in a container created from this image, and also sets some parameters for that command. Rebuild your image:

```
docker image build -t <username>/myimage:1.0 .
```

Run a container from your new image with no command provided:

```
docker container run <username>/myimage:1.0
```

You should see the command provided by the CMD parameter in the Dockerfile running. Try explicitly providing a command when running a container:

```
docker container run <username>/myimage:1.0 echo "hello world"
```

Providing a command in docker container run overrides the command defined by CMD.

7.2 Default Commands via ENTRYPOINT

Replace the CMD instruction in your Dockerfile with an ENTRYPOINT:

```
ENTRYPOINT ["ping"]
```

Build the image and use it to run a container with no process arguments:

```
docker image build -t <username>/myimage:1.0 .
docker container run <username>/myimage:1.0
```

What went wrong? Try running with an argument after the image name:

```
docker container run <username>/myimage:1.0 127.0.0.1
```

Tokens provided after an image name are sent as arguments to the command specified by ENTRYPOINT.

7.3 CMD and ENTRYPOINT Together

Open your Dockerfile and modify the ENTRYPOINT instruction to include 2 arguments for the ping command:

```
ENTRYPOINT ["ping", "-c", "3"]
```

If CMD and ENTRYPOINT are both specified in a Dockerfile, tokens listed in CMD are used as default parameters for the ENTRYPOINT command. Add a CMD with a default IP to ping:

```
CMD ["127.0.0.1"]
```

Build the image and run a container with a public IP as an argument to `docker container run`:

```
docker container run <username>/myimage:1.0 <some public ip>
```

Run another container without any arguments after the image name. Explain the difference in behavior between these two last containers.

7.4 Conclusion

In this exercise, we encountered the Dockerfile commands CMD and ENTRYPOINT. These are useful for defining the default process to run as PID 1 inside the container right in the Dockerfile, making our containers more like executables and adding clarity to exactly what process was meant to run in a given image's containers.

8 Dockerizing an Application

8.1 Setting up a Java App

For an alternative way of doing this see below (Optional Exercise)

Install Java 8:

```
sudo apt-get install openjdk-8-jdk
```

Create a directory called `javahelloworld`; in that directory, make a file called `HelloWorld.java`, containing the following code:

```
public class HelloWorld
{
    public static void main (String [] args)
    {
        System.out.println("hello world");
    }
}
```

Compile your new application:

```
javac HelloWorld.java
```

And run it:

```
java HelloWorld
```

If all's gone well, you have a hello world program running in Java; next, we'd like to containerize this application by capturing its environment in an image described by a Dockerfile.

8.2 Dockerize your App

In your `javahelloworld` folder, create a Dockerfile that bases its image off of the `java:8` base image:

```
FROM java:8
```

Add your source code into your image using a new Dockerfile command, `COPY`:

```
...
COPY HelloWorld.java /
```

`COPY`'s syntax is `COPY <target> <destination>`, which copies files from the build context into the image. If `<target>` is a directory, all the contents of that directory will be copied to `<destination>`.

Compile your app in the image by appending to your Dockerfile:

```
...
RUN javac HelloWorld.java
```

Use `ENTRYPOINT` to run your application automatically when a container is launched from this image:

```
...
ENTRYPOINT ["java", "HelloWorld"]
```

Build the image, use it to run a container running the default `ENTRYPOINT` command, and observe the output.

8.3 Restructure your Application

After running a container from your image with the default command, try overriding the `ENTRYPOINT` command with `--entrypoint`; we'll run `bash`, so we can explore our containerized environment interactively with a bash shell.

```
docker container run -it --entrypoint bash <your java image ID>
```

Find where the `HelloWorld.java` source is, and where the compiled binary is. We can restructure our app to be a little more organized as follows; back on your host machine under the `javahelloworld` directory, make two subdirectories `src` and `bin`.

Put your java source into this src directory, and recompile so the binary ends up in the bin directory:

```
javac -d bin src/HelloWorld.java
```

Run the application again with java -cp bin HelloWorld to make sure everything is still working; if it is, we're ready to modify our Dockerfile to reflect this organizational structure in our image. Modify the COPY instruction to copy all files in the src folder on your host into /home/root/javahelloworld/src in your image:

```
...
COPY src /home/root/javahelloworld/src
```

Modify the RUN instruction to compile the code by referencing the correct src folder and to place the compiled code into the bin folder:

```
...
RUN javac -d bin src/HelloWorld.java
```

Modify ENTRYPOINT to specify java -cp bin :

```
...
ENTRYPOINT ["java", "-cp", "bin", "HelloWorld"]
```

Try building an running your image now; you should get an error since your source code is now living under /home/root/javahelloworld/src, and the RUN command told the compiler to go looking under ./src. By default, commands are run at the root of the image's filesystem, but we can modify this with the WORKDIR Dockerfile instruction, which sets the position in the filesystem for all subsequent commands; before the RUN instruction, add:

```
WORKDIR /home/root/javahelloworld
```

Try building your image and running the container with the default ENTRYPOINT command again. There's still an error, but a different one; modify your Dockerfile to fix this, rebuild your image and verify that everything is working correctly again.

Finally, override the default ENTRYPOINT command and run bash instead, like we did above. Verify that the directory structure you described in your Dockerfile is reflected in your image's filesystem.

8.4 Conclusion

In this exercise, you dockerized a simple application, and learned how to manipulate and navigate an image's filesystem with the Dockerfile commands COPY and WORKDIR. With some practice, writing a Dockerfile that builds up an application image follows a very similar pattern to setting up the app natively, expressing things with RUN, COPY, WORKDIR and ENTRYPOINT commands.

8.5 Optional Exercise

For an alternative way to create the Java application without having to install Java on the host you can use a Java container instead. Run a Java container in interactive mode

```
docker container run --rm -it java:8 bash
```

Install an editor inside the container, either VIM or nano

```
apt-get update && apt-get install -y vim
# or install nano
# apt-get update && apt-get install -y nano
```

Now still inside the container create a directory `javahelloworld` and create the file `HelloWorld.java` in it using the editor you just installed, e.g. with VIM that would be

```
mkdir javahelloworld
cd javahelloworld
vim HelloWorld.java
```

add the code to this file, save and quit the editor. Still inside the Java container compile and run the application

```
javac HelloWorld.java
java HelloWorld
```

9 Managing Images

9.1 Tagging and Listing Images

Download the `ubuntu:14.04` image from Docker Hub:

```
docker image pull ubuntu:14.04
```

Make a new tag of this image:

```
docker image tag ubuntu:14.04 my-ubuntu:dev
```

List your images:

```
docker image ls
```

You should have `ubuntu:14.04` and `my-ubuntu:dev` both listed, but they ought to have the same hash under image ID, since they're actually the same image.

9.2 Sharing Images on Dockerhub

Next, let's share our image on Docker Hub. If you don't already have an account there, head over to hub.docker.com and make one.

Push your image to Docker Hub:

```
docker image push my-ubuntu:dev
```

You should get an `authentication required` error. Login by doing `docker login`, and try pushing again. The push fails again because we haven't namespaced our image correctly for distribution on Docker Hub; all images you want to share on Docker Hub must be named like `<your Docker Hub username>/<repo name>[:<optional tag>]`.

Retag your image to be namespaced properly, and push again:

```
docker image tag my-ubuntu:dev <dockerhub username>/my-ubuntu:dev
docker image push <dockerhub username>/my-ubuntu:dev
```

Visit your Docker Hub page, find your new `my-ubuntu` repo, and confirm that you can see the `:dev` tag therein. Explore your new repo, and fill out the description and other fields you find there.

Next, write a Dockerfile that uses `<dockerhub username>/my-ubuntu:dev` as its base image, and installs any application you like on top of that. Build the image, and simultaneously tag it as `:1.0`:

```
docker image build -t <dockerhub username>/my-ubuntu:1.0 .
```

Push your `:1.0` tag to Dockerhub, and confirm you can see it in the appropriate repo.

Finally, list the images currently on your node with `docker image ls`. You should still have the version of your image that wasn't namespaced with your Docker Hub user name; delete this using `docker image rm`:

```
docker image rm my-ubuntu:dev
```

9.3 Private Images

Explore the UI on your Docker Hub repo `<username>/my-ubuntu`, and find the toggle to make that repo private.

Pair up with someone sitting next to you, and try to pull their image:

```
docker image pull <partner's dockerhub name>/my-ubuntu:1.0
```

Their private repo should be invisible to you at this time. Add each other as Collaborators to your `my-ubuntu` repo, and pull again; if all has gone well, you should now be able to pull their repo. Also check to see that you can see each other's repo on your 'Repositories' tab of your Docker Hub profile.

9.4 Conclusion

In this exercise, we saw how to name and tag images with `docker image tag`; explored the correct naming conventions necessary for pushing images to your Docker Hub account with `docker image push`; learned how to remove old images with docker image rm; and learned how to make repos private and share them with collaborators on Docker Hub.

10 Cleanup Commands

Once we start to use Docker heavily on our system we want to understand what resouces the Docker engine is using. We also want ways on how we can cleanup and free unused resources. For all this we can use the `docker system` commands.

Let's find out how much memory Docker is using by executing:

```
docker system df
```

The output will show us how much space images, containers and (local) volumes are occupying and how much of this space can be reclaimed. The output might look similar to this

```
TYPE           TOTAL          ACTIVE         SIZE           RECLAIMABLE
Images          50             2             9.012 GB      7.271 GB (80%)
Containers        2             2             69.29 MB       0 B (0%)
Local Volumes     0               0               0 B            0 B
```

Now let's reclaim all reclaimable space by using the following command:

```
docker system prune
```

And we should be asked if we really want to remove all unused networks, containers, images and volumes. Answer with `y`.

This is the most radical way of keeping our system clean. If we want to be more focused and only cleanup resources of a given type we can use the following commands:

```
docker image prune
docker container prune
docker volume prune
docker network prune
```

Try each of them out. If you are tired of answering `y` each time you can add the flag `-f` or `--force` to the command, e.g.:

```
docker system prune --force
```

This is especially useful if you want to run the above commands as part or an automation script.

11 Inspection Commands

11.1 System Information

We can find the `info` command under `system`. Execute:

```
docker system info
```

and analyze the output. It gives us plenty of information about our Docker host.

When looking at the output, can you identify:

- how many images are cached on your machine?
- how many containers are running or stopped?
- what version of containerd are you running?
- whether Docker is running in swarm mode?
- etc.

11.2 System Events

There is another powerful system command that allows us to monitor what's happening on the Docker host. Execute the following command:

```
docker system events
```

Please note that it looks like the system is hanging, but that is not the case. The system is just waiting for some events to happen.

Open a second terminal and execute the following command:

```
docker container run --rm alpine echo 'Hello World!'
```

and observe the generated output in the first terminal. It should look similar to this:

```
2017-01-25T16:57:48.553596179-06:00 container create 30eb630790d44052f26c1081dbfe8636537da911
2017-01-25T16:57:48.556718161-06:00 container attach 30eb630790d44052f26c1081dbfe8636537da911
2017-01-25T16:57:48.698190608-06:00 network connect de1b2b40f522e69318847ada36958e32a1ca47d7f
2017-01-25T16:57:49.062631155-06:00 container start 30eb630790d44052f26c1081dbfe8636537da9119
2017-01-25T16:57:49.065552570-06:00 container resize 30eb630790d44052f26c1081dbfe8636537da911
2017-01-25T16:57:49.164526268-06:00 container die 30eb630790d44052f26c1081dbfe8636537da911991
2017-01-25T16:57:49.613422740-06:00 network disconnect de1b2b40f522e69318847ada36958e32a1ca47
2017-01-25T16:57:49.815845051-06:00 container destroy 30eb630790d44052f26c1081dbfe8636537da91
```

If you don't like the format of the output then we can use the `--format` parameter to define our own format in the form of a Go template. Stop the events watch on your first terminal with `CTRL+C`, and try this:

```
docker system events --format '--> {{.Type}}-{{.Action}}'
```

now the output looks a little bit less cluttered when we run our alpine container on the second terminal as above:

```
--> container-create
--> container-attach
--> network-connect
--> container-start
--> container-resize
--> container-die
--> network-disconnect
--> container-destroy
```

Finally we can find out what the event structure looks like by outputting the events in `json` format (once again after killing the events watcher on the first terminal and restarting it with):

```
docker events --format '{{json .}}'
```

which should give us for the first event in the series after re-running our alpine container on the second node something like this (note, the output has been prettyfied for readability):

```
{
   "status":"create",
  "id":"95ddb6ed4c87d67fa98c3e63397e573a23786046e00c2c68a5bcb9df4c17635c",
   "from":"alpine",
   "Type":"container",
   "Action":"create",
   "Actor":{
    "ID":"95ddb6ed4c87d67fa98c3e63397e573a23786046e00c2c68a5bcb9df4c17635c",
      "Attributes":{
         "image":"alpine",
         "name":"sleepy_roentgen"
      }
   },
   "time":1485385702,
   "timeNano":1485385702748011034
}
```

11.3 Summary

In this lab we have learned how to inspect system wide properties of our Docker host by usign the `docker system inspect` command. We have also used the `docker system events` command to further inspect the activity on the system when containers and other resources are created, used and destroyed.

12 Image Autobuilds

12.1 Setting up a Code Repo

Start by creating a new public GitHub repository called `javahelloworld`.

Recreate the Dockerfile and java source for our java hello world example in a clean directory:

./Dockerfile:

```
FROM java:7

COPY src /home/root/javahelloworld/src
WORKDIR /home/root/javahelloworld
RUN mkdir bin
RUN  javac -d bin src/HelloWorld.java
ENTRYPOINT ["java", "-cp", "bin", "HelloWorld"]
```

./src/HelloWorld.java:

```
public class HelloWorld
{
    public static void main (String [] args)
    {
        System.out.println("hello world");
    }
}
```

Initialize this directory as a git repo, point it at your new GitHub repo, and push:

```
git init
git remote add origin <your GitHub repo>
git add *
git commit -m 'first post'
git push origin master
```

12.2 Starting Automated Builds

Next, we need to use the Docker Hub UI to set up automatic image building based on the Dockerfile in our GitHub repo. Log into Docker Hub, and from the “Create” menu at the top of the page, choose “Create Automated Build”. If you’ve never done this before, you’ll be prompted to link your Docker Hub account with your GitHub or Bitbucket account. Do so, then click “Create Automated Build” again, and select your `javahelloworld` repository. Leave all the defaults as is.

Finally, trigger yor first build by navigating through “Build Settings” -> “Trigger Build” on the Docker Hub page corresponding to your new image, at `<docker hub user name>/javahelloworld`. Investigate the details of this build on the “Build Details” page.

12.3 Automated Build in Action

Change the source in `./src/HelloWorld.java` to print something other than `Hello World`. After verifying that things work locally, commit your changes and push to GitHub:

```
git commit -a -m 'new message'
git push origin master
```

Back on Docker Hub, visit your repo's "Build Details" tab; you should see a new build that started automatically when you pushed to GitHub. Find the new image's ID, and use this to pull the new image down locally. Run the image and verify that it returns the new message you just committed.

12.4 Conclusion

In this exercise, you set up a Docker Hub repo to automatically build an image based on a Dockerfile committed to GitHub. In a CI pipeline, a testing server could then pull in this image, verify that tests all pass, and then pass the image to production. Because the code always runs in a container defined by the Dockerfile provided by the application developer, the application is guaranteed to run the same way across all environments in your pipeline.

13 Creating and Mounting Volumes (1/3)

13.1 Creating a Volume

Create a volume called test1:

```
docker volume create --name test1
```

Run `docker volume ls` and verify that you can see your `test1` volume.

Execute a new `ubuntu` container and mount the `test1` volume. Map it to the path `/www/website` and run bash as your process:

```
docker container run -it -v test1:/www/website ubuntu:14.04 bash
```

Inside the container, verify that you can get to `/www/website`:

```
cd /www/website
```

Create a file called `test.txt` inside the `/www/website` folder:

```
touch test.txt
```

Exit the container without stopping it by hitting `CTRL + P + Q`.

13.2 Volumes During Image Creation

Commit the updated container as a new image called test and tag it as `1.0`:

```
docker container commit <container ID> test:1.0
```

Execute a new container with your test image and go into its bash shell:

```
docker container run -it test:1.0 bash
```

Verify that the /www/website folder exists. Are there any files inside it? Exit this container.

Run docker container ls to ensure that your first container is still running.

13.3 Conclusion

In this exercise, we saw how to create and mount a volume, and add data to it from within a container. We also saw that making an image out of a running container via docker container commit does *not* capture any information from volumes mounted inside the container; volumes and images are created and updated completely independently.

14 Creating and Mounting Volumes (2/3)

14.1 Finding the Host Mountpoint

Run docker volume inspect on the test1 volume to find out where it is mounted on the host machine:

```
docker volume inspect test1
```

Copy the path specified by the Mountpoint field. The path should be /var/lib/docker/volumes/test1/_data:

Elevate your user privileges to root:

```
sudo su
```

Change directory into the volume path in step 2:

```
cd /var/lib/docker/volumes/test1/_data
```

Run ls and verify you can see the test.txt file you created inside your original container.

Create another file called test2.text inside this directory:

```
touch test2.txt
```

Exit the superuser account:

```
exit
```

Use docker container exec to log back into the shell of your ubuntu container that is still running:

```
docker container exec -it <container name> bash
```

Change directory into the /www/website folder, and verify that you can see both the test.txt and test2.txt files.

14.2 Conclusion

In this exercise, we saw the `docker volume inspect` command, and explored how to use it to identify where volume data is being written on the host machine.

15 Creating and Mounting Volumes (3/3)

15.1 Deleting Volumes

After exiting yor container and returning to the host's bash prompt, delete the remaining container without using any options (you may need to stop the container first if it's still running):

```
docker container rm <container ID>
```

Run `docker volume ls` and check the result; notice our `test1` volume is still present, since removing containers doesn't affect mounted containers.

Check to see that the `test.txt` and `test2.txt` files are also still present in your volume on the host:

```
sudo ls /var/lib/docker/volumes/test1/_data
```

Delete the `test1` volume:

```
docker volume rm test1
```

Run `docker volume ls` and make sure the `test1` volume is in fact gone.

15.2 Conclusion

In this exercise, we saw how to remove volumes with `docker volume rm`.

16 Mounting Host Directories

On your host, make a directory called 'demo' and add a file to it:

```
host> mkdir demo
host> touch demo/file1.dat
```

Mount 'demo' in a container:

```
host> docker container run -it -v demo:/www/demo ubuntu:14.04 bash
```

In the container, 'ls' the '/www/demo' directory. Do you see 'file1.dat'?

Add another file to '/www/demo' from within the container:

```
container> touch /www/demo/file2.dat
```

Exit the container using CTRL+P+Q. What is in the 'demo' directory you made in step 1? Where are file1.dat and file2.dat?

Modify file1.dat on your host, run a new container from within the demo directory you made via:

```
host> docker container run -it -v ${PWD}:/www/demo ubuntu:14.04 bash
```

and verify that the changes to file1.dat appear inside the container.

16.1 Conclusion

In this exercise, you saw how to mount a directory on your host machine inside a running container, with a twist: absolute paths (like what gets returned by ${PWD} in the final example) mount host directories as advertised, but non-absolute paths get interpreted as volume names and are created or mounted as such, which is what happened in the first docker run - instead of mounting the demo directory we wanted, the non-absolute path was interpreted as a volume, which was created and mounted in the container.

17 Volumes Usecase: Recording Logs

17.1 Set up an app with logging

Create a volume called nginx_logs:

```
host> docker volume create --name nginx_logs
```

Run the custom trainingteam/nginx container and map your public_html host folder to a directory at /usr/share/nginx/html. Also mount your nginx_logs volume to the /var/log/nginx folder. Name the container nginx_server:

```
host> docker container run -d -P --name nginx_server \
        -v ~/public_html:/usr/share/nginx/html \
        -v nginx_logs:/var/log/nginx \
        trainingteam/nginx
```

Get terminal access to your container:

```
host> docker container exec -it nginx_server bash
```

Put some text into /usr/share/nginx/html/index.html, and then exit the terminal.

Run docker container ls to find the host port which is mapped to port 80 on the container. In your browser, access the URL and port nginx is exposed on.

Verify you can see the contents of your index.html file from your public_html folder on your host.

17.2 Inspect Logs on the Host

Get terminal access to your container again:

```
host> docker container exec -it nginx_server bash
```

Change directory to `/var/log/nginx`:

```
container> cd /var/log/nginx
```

Check that you can see the `access.log` and `error.log` files.

Run `tail -f access.log`, refresh your browser a few times and observe the log entries being written to the file. Exit the container terminal after seing a few live log entries.

Run `docker volume inspect nginx_logs` and copy the path indicated by the "Mountpoint" field; path should be `/var/lib/docker/volumes/nginx_logs/_data`.

Check for the presence of the `access.log` and `error.log` files, then follow the tail of `access.log`:

```
host> sudo ls /var/lib/docker/volumes/nginx_logs/_data
host> sudo tail -f /var/lib/docker/volumes/nginx_logs/_data/access.log
```

Refresh your browser a few times in order to make some requests to the NGINX server; observe log entries being written into the `access.log` file, available in the `nginx_logs` volume on your host machine.

17.3 Conclusion

In this exercise, you explored how mounting volumes makes live data being generated inside a container available to other containers and the outside world; the information nginx was writing to the logging volume can be consumed in real time by independent monitoring applications that would survive the failure or deletion of the nginx container.

18 Sharing Volumes

Run `docker container ls` and make sure that your `nginx_server` container from the last step is still running, otherwise run it again with this command:

```
host> docker container run -d -P --name nginx_server \
        -v ~/public_html:/usr/share/nginx/html \
        -v nginx_logs:/var/log/nginx \
        trainingteam/nginx
```

Run a new `ubuntu` container and mount the `nginx_logs` volume to the folder `/data/mylogs` as read only, with bash as your process:

```
host> docker container run -it \
        -v nginx_logs:/data/mylogs:ro \
        ubuntu:14.04 bash
```

In your new container's terminal, change directory to `/data/mylogs`

Confirm that you can see the `access.log` and `error.log` files.

Try and create a new file called `text.txt`

```
container> touch test.txt
```

Notice how it fails because we mounted the volume as read only.

18.1 Conclusion

In this exercise, you saw how to mount a pre-existing volume inside a new container; the syntax is nominally exactly the same as for mounting a new container, but in this case we also used the `:ro` flag to mount that volume in read-only mode, so the new container can't damage any data in the volume. This is also an important security best practice, since sharing volumes between containers is a way to break isolation between containers; by allowing read-only access, we prevent the new container from injecting any malicious data into the volume that would then appear inside all other containers using that volume, as well as the host.

19 Docker Plugins

Plugins are used to extend the capabilities of the Docker Engine. Anyone, not just Docker, can implement plugins. Currently only volume driver plugins are supported, but in future support for more types will be added.

19.1 Installing a Plugin

Plugins can be hosted on Docker Hub or any other (private) repository. Let's start with Docker Hub. Browse to hub.docker.com and enter `vieux/sshfs` in the search box. The result should show you the plugin that we are going to work with. Notice in the **Docker Pull Command** section that it appears that we can actually pull a plugin like a normal Docker image:

```
docker pull vieux/sshfs
```

execute the above command in your terminal. It should result in an error. It is currently not possible to `pull` plugins. A plugin is implicitly pulled from the repositroy when we install it.

Now let's try to install the plugin into our Docker Engine by using this command:

```
docker plugin install vieux/sshfs
```

The system should ask us for permission to use privileges. In the case of the sshfs plugin those are 3 permissions. Answer with y.

Once we have successfully installed some plugins we can use the ls command to see the status of each of the installed plugins. Execute:

```
docker plugin ls
```

19.2 Enabling and Disabling a Plugin

Once a plugin is installed it is enabled by default. We can disable it using this command:

```
docker plugin disable vieux/sshfs
```

only when a plugin is disabled can certain operations on it be executed.

The plugin can be (re-) enabled by using this command:

```
docker plugin enable vieux/sshfs
```

Play with the above commands and notice how the status of the plugin changes when displaying it with docker plugin ls.

19.3 Inspecting a Plugin

We can also use the inspect command to further inspect all the attributes of a given plugin. Execute the following command:

```
docker plugin inspect vieux/sshfs
```

and analyze the output. Specifically note that there are two sections in the metadata called Env, one is under Config and the other under Settings. This is where the list of environment variables are listed that the author of the plugin has defined. In this specific situation we can see that there is a single variable called DEBUG defined. Its initial value is 0.

We can use the set command to change values of the environment variables. Execute:

```
docker plugin set vieux/sshfs DEBUG=1
```

and then inspect again the metadata of the plugin. Notice how the value of DEBUG has been adjusted. Only the one under the Settings node changed but the one under the Config node still shows the original (default) value. Please note that the above command can only be executed if the plugin has been disabled first.

We could also have defined the value of the environment variables during installation of the plugin, namely:

```
docker plugin install vieux/sshfs DEBUG=1
```

19.4 Removing a Plugin

If we don't want or need this plugin anymore we can remove it using the command:

```
docker plugin disable vieux/ssfs
docker plugin rm vieux/sshfs
```

Note how we first have to disable the plugin before we can remove it.

19.5 Using the Plugin

> **Note:** This exercise requires that you have access to a folder on a remote host which you can access via SSH with username and password. This can either be a host provided to you by your trainer or your own host if you have any. Unfortunately at this time the plugin does not work with public/private key access but only with username/password.

To use the plugin we create a Docker volume like this:

```
docker volume create -d vieux/sshfs \
    -o sshcmd=<user@host:path> \
    -o password=<password> \
    sshvolume
```

replace user, host, path and password by values provided to you by the trainer, or if you have your own remote server you can SSH into with username/password then you can use that one.

Now we can use that volume to access the remote folder and work with it as follows. Execute the following command to run an alpine container which has access to the remote volume:

```
docker container run --rm -it -v sshvolume:/data alpine sh
```

inside the container navigate to the /data folder and create a new file:

```
cd /data
echo 'Hello from client!' > <your-name>.txt
ls -al
```

Start a second container like this:

```
docker container run --rm -it -v sshvolume:/shared busybox ls -al /shared
```

and verify that the previously generated file <your-name>.txt is present.

19.6 Summary

In this task we have learned how to install, inspect and remove plugins into or from a Docker engine. We have specifically installed a plugin that allows us to access a remote volume via SSH. We have then created a Docker volume that uses this plugin and have demonstrated it's usage.

20 Introduction to Container Networking

20.1 The Default Bridge Network

First, let's investigate the linux bridge that Docker provides by default. Start by installing bridge utilities:

```
apt-get install bridge-utils
```

Ask for information about the default Docker linux brdige, `Docker0`:

```
brctl show docker0
```

Start some named containers and check again:

```
docker container run --name=u1 -dt ubuntu:14.04
docker container run --name=u2 -dt ubuntu:14.04
brctl show docker0
```

You should see two new some virtual ethernet (veth) connections to the bridge, one for each container. Veth connections are a linux feature for creating an access point to a sandboxed network namespace.

The `docker network inspect` command yields network information about what containers are connected to the specified network; the default network is always called `bridge`, so run:

```
docker network inspect bridge
```

and find the IP of your container `u1`.

Next, connect to container `u2` of your containers using `docker container exec -it u2 /bin/bash`. From here, try pinging container `u1` by the IP address you found in the previous step; then try pinging `u1` by container name, `ping u1` - notice the lookup works with the IP, but not with the container name in this case.

Still inside container `u2`, run `ip a` to see some information about what the network connection looks like from inside the container. Find the `eth0` entry, and confirm that the MAC address and IP assigned are the same (Docker always assigns MAC and IP pairs in this way, to avoid collisions).

Finally, back on the host, run `docker container inspect u2`, and look for the `NetworkSettings` key to see what this connection looks like from outside the container's network namespace.

20.2 User Defined Bridge Networks

In the last step, we investigated the default bridge network; now let's try making our own. User defined bridge networks work exactly the same as the default one, but provide DNS lookup by container name, and are firewalled from other networks by default.

Create a bridge network by using the `bridge` driver with `docker network create`:

```
docker network create --driver bridge my_bridge
```

Examine what networks are available on your host:

```
docker network ls
```

You should see `my_bridge` and `bridge`, the two bridge networks, as well as `none` and `host` - these are two other default networks that provide no network stack or connect to the host's network stack, respectively.

Launch a container connected to your new network via the `--network` flag:

```
docker container run --name=u3 --network=my_bridge -dt ubuntu:14.04
```

Use the `inspect` command to investigate the network settings of this container:

```
docker container inspect u3
```

`my_bridge` should be listed under the `Networks` key. Launch another container, this time interactively:

```
docker container run --name=u4 --network=my_bridge -it ubuntu:14.04
```

From inside container `u4`, ping `u3` by name: `ping u3`. Recall this didn't work on the default bridge network between `u1` and `u2`; DNS lookup by container name is only enabled for explicitly created networks.

Finally, try pinging `u1` by IP or container name as you did in the previous step, this time from container `u4`. `u1` (and `u2`) are not reachable from `u4` (or `u3`), since they reside on different networks; all Docker networks are firewalled from each other by default.

20.3 Inter-Container Communication

In the last step, we saw how containers living on separate networks are firewalled from each other; this is a fundamental security feature of container networking, and should be leveraged as much as possible; if two containers don't need to talk to each other, put them on separate Docker networks, exactly as you saw in the last example.

However, in real circumstances there may be complicated communication requirements between containers, which permit isolation between containers A and B, but requires connectivity for both to container C. As such, Docker supports connecting containers to more than one network, to create an expressive framework for communication channels between containers.

Recall your container u2 is currently plugged in only to the default bridge network; confirm this using docker container inspect u2. Connect u2 to the my_bridge network:

```
docker network connect my_bridge u2
```

Check that you can ping the u3 and u4 containers from u2:

```
docker container exec u2 ping u3
docker container exec u2 ping u4
```

Check that you can ping the u2 and u4 container from u3

```
docker container exec u3 ping u2
docker container exec u3 ping u4
```

But note u1 still can't reach u3 and u4:

```
docker container exec u1 ping u3
docker container exec u1 ping u4
```

20.4 Conclusion

In this exercise, you explored the fundamentals of container networking. The key take away is that *containers on separate networks are firewalled from each other by default*. You also explored a number of API objects:

- docker network ls lists all networks on the host
- docker network inspect <network name> gives more detailed info about the named network
- docker network create --driver <driver> <network name> creates a new network using the specified driver; so far, we've only seen the bridge driver, for creating a linux bridge based network.
- docker network connect <network name> <container name or id> connects the specified container to the specified network after the container is running; the --network flag in docker container run achieves the same result at container launch.
- docker container inspect <container name or id> yields, among other things, information about the networks the specified container is connected to.

21 Container Port Mapping

21.1 Port Mapping at Runtime

Run an nginx container with no special port mappings:

```
docker container run -d nginx
```

nginx stands up a landing page at `<ip>:80`; try to visit this at your host or container's IP, and it won't be visible; no external traffic can make it past the linux bridge's firewall to the nginx container.

Now run an nginx container and map port 80 on the container to port 5000 on your host using the `-p` flag; also map port 8080 on the container to port 9000 on the host:

```
docker container run -d -p 5000:80 -p 9000:8080 nginx
```

Note that the syntax is: `-p [host-port]:[container-port]`. Verify the port mappings with the `docker container port` command

```
docker container port <container id>
```

Visit your nginx landing page at `<host ip>:5000`.

21.2 Exposing Ports from the Dockefile

In addition to manual port mapping, we can expose some ports in a Dockerfile for automatic port mapping on container startup. In a fresh directory, create a Dockerfile:

```
FROM nginx

EXPOSE 80 8080
```

Build your image with `docker image build -t my_nginx .`. Use the `-P` flag when running to map all ports mentioned in the `EXPOSE` directive:

```
docker container run -d -P my_nginx
```

Use `docker container ls` or `docker container port` to find out what host ports were used, and visit your nginx landing page at the appropriate ip/port.

21.3 Conclusion

In this exercise, we saw how to explicitly map ports from our container's network stack onto ports of our host at runtime with the `-p` option to `docker container run`, or more flexibly in our Dockerfile with `EXPOSE`, which will result in the listed ports inside our container being mapped to random available ports on our host.

22 Starting a Compose App

In a microservice-oriented design pattern, labor is divided among modular, independent services, many of which cooperate to form a full application. Docker images and containerization naturally enable this paradigm by using images to define services, and containers to correspond to instances of those services. In order to be successful, each running container will need to be able to interact; Docker Compose facilitates these interactions on a single host. In this example, we'll explore a toy example of such an application orchestrated by Docker Compose.

22.1 Prepare Service Images

Download the Dockercoins app from github:

```
git clone https://github.com/docker-training/orchestration-workshop.git
cd orchestration-workshop
git fetch origin 17.3
git checkout 17.3
```

This app consists of 5 services: a random number generator `rng`, a hasher, a backend worker, a redis queue, and a web frontend. Each service has a corresponding image, and to begin with, we'll run a single container for each, orchestrated by Docker Compose. It's possible to have Docker Compose build images on the fly, but it's better to use pre-built images available in a registry, like Docker Hub; that way, your app is portable across machines. Log in to your Docker Hub account from the command line now, using `docker login` (if you don't have a Docker Hub account, make a free one first at hub.docker.com).

Each custom image lives in its own directory; build each tagged as `<your dockerhub id>/dockercoins_<service name>:1.0`, and push to Docker Hub. For `hasher`, this looks like:

```
cd dockercoins
docker image build -t <username>/dockercoins_hasher:1.0 hasher
docker image push <username>/dockercoins_hasher:1.0
```

Do the exact same thing for `rng`, `worker` and `webui`. Have a look in `docker-compose.yml`, and change all the `image` values to have your Docker Hub id instead of `user`; now you'll be able to use this Compose file to set up your app on any machine that can reach Docker Hub.

22.2 Start the App

Stand up the app (you may need to install Docker Compose first, if this didn't come pre-installed on your machine; see the instructions at https://docs.docker.com/compose/install/):

```
docker-compose up
```

Logs from all the running services are sent to STDOUT. Let's send this to the background instead; kill the app with `CTRL+C`, sending a `SIGTERM` to all running processes; some exit immediately, while others wait for a 10s timeout before being killed by a subsequent `SIGKILL`. Start the app again in the background:

```
docker-compose up -d
```

Now check out which containers are running thanks to Compose:

```
docker-compose ps
```

Compare this to the usual `docker container ls`; at this point, it should look about the same. Start any other container with `docker container run`, and try both `ls` commands again. Do you notice any difference?

With all five containers running, visit Dockercoins' web UI at :8000. You should see a chart of mining speed, around 4 hashes per second.

22.3 Viewing Logs

See logs from a Compose-managed app via:

```
docker-compose logs
```

The logging API in Compose follows the main Docker logging API closely. For example, try following the tail of the logs just like you would for regular container logs:

```
docker-compose logs --tail 10 --follow
```

Note that when following a log, `CTRL+S` and `CTRL+Q` pauses and resumes live following.

22.4 Conclusion

In this exercise, you saw how to start a pre-defined Compose app, and how to inspect its logs.

23 Scaling a Compose App

23.1 Scaling a Service

Any service defined in our `docker-compose.yml` can be scaled up from the Compose API; in this context, 'scaling' means launching multiple containers for the same service, which Docker Compose can route requests to and from. Scale up the `worker` service in our Dockercoins app to have two workers generating coin candidates, while checking the list of running containers before and after:

```
docker-compose ps
docker-compose scale worker=2
docker-compose ps
```

Look at the performance graph provided by the web frontend; the coin mining rate should have doubled. Also check the logs using the logging API we learned in the last exercise; you should see a second `worker` instance reporting.

23.2 Scale Nonlinearity

Try running `top` to inspect the system resource usage; it should still be fairly neglegible. So, keep scaling up your workers:

```
docker-compose scale worker=10
docker-compose ps
```

Check your web frontend again; has going from 2 to 10 workers provided a 5x performance increase? It seems that something else is bottlenecking our application; any distributed application such as Dockercoins needs tooling to understand where the bottlenecks are, so that the application can be scaled intelligently.

Look in `docker-compose.yml` at the `rng` and `hasher` services; they're exposed on host ports 8001 and 8002, so we can use `httping` to probe their latency:

```
httping -c 10 localhost:8001
httping -c 10 localhost:8002
```

`rng` on port 8001 has the much higher latency, suggesting that it might be our bottleneck. A random number generator based on entropy won't get any better by starting more instances on the same machine; we'll need a way to bring more nodes into our application to scale past this, which we'll explore in the next unit on Docker Swarm. For now, shut your app down:

```
docker-compose down
```

23.3 Conclusion

In this exercise, we saw how to scale up a service defined in our Compose app using the `scale` API object. Also, we saw how crucial it is to have detailed monitoring and tooling in a microservices-oriented application, in order to correctly identify bottlenecks and take advantage of the simplicity of scaling with Docker.

24 Creating a Swarm

In this exercise, we'll see how to set up a swarm using Docker Swarm Mode, including joining workers and promoting workers into the manager consensus.

24.1 Start Swarm Mode

Enable Swarm Mode on whatever node is to be your first manager node:

```
docker swarm init
```

Try doing

```
docker info
```

to see a confirmation that Swarm Mode is active along with some other information about the swarm. Finally, try

```
docker node ls
```

A single node is reported in the cluster.

24.2 Add Workers to the Swarm

A single node swarm is not a particularly interesting swarm; let's add some workers to really see Swarm Mode in action. Swarm Mode will report the cut-and-paste command to join a new node to the swarm as a worker via the following, executed on a manger node:

```
docker swarm join-token worker
```

SSH into a second node, and paste the result of this command there. This new node will have joined the swarm as a worker. Do `docker node ls` on the manager again, and you should see both your nodes and their status; note that `docker node ls` won't work on a worker node, as the cluster status is maintained only by the manager nodes.

Finally, use the same join token to add two more workers to your swarm. When you're done, confirm that `docker node ls` on your one manager node reports 4 nodes in the cluster - one manager, and three workers.

24.3 Promoting Workers to Managers

At this point, our swarm has a single manager. If this node goes down, the whole Swarm is lost. In a real deployment, this is unacceptable; we need some redundancy to our system, and Swarm Mode achieves this by allowing a Raft consensus of multiple managers to preserve swarm state. Promote two of your workers to manager status by executing, on the current manager node:

```
docker node promote worker-0 worker-1
```

where `worker-0` and `worker-1` are the hostnames of the two workers you want to promote to managerial status (look at the output of `docker node ls` if you're not sure what your hostnames are). Finally, do a `docker node ls` to check and see that you now have three managers. Note that manager nodes also count as worker nodes - tasks can still be scheduled on them as normal.

24.4 Conclusion

In this exercise, you saw how to set up a swarm with with basic API objects `docker swarm init` and `docker swarm join`, as well as how to inspect the state of the swarm with `docker node ls` and `docker info`. Finally, you promoted worker nodes to the manager consensus with `docker node promote`.

25 Starting a Service

So far, we've set up a four-node swarm with three managers; in order to use a swarm to actually execute anything, we have to define *services* for that swarm to run; services are the fundamental logical entity that users interact with in a distributed application engineering environment, while things like individual processes or containers are handled by the swarm scheduler; similarly, the scheduler handles routing tasks to specific nodes, so the user can approach the swarm as a whole without explicitly interacting with individual nodes.

25.1 Start a Service

Create a service featuring an `alpine` container pinging Google resolvers:

```
docker service create alpine ping 8.8.8.8
```

Note the syntax is a lot like `docker container run`; an image (`alpine`) is specified, followed by the PID 1 process for that container (`ping 8.8.8.8`). Get some information about the currently running services:

```
docker service ls
```

Check which node the container was created on:

```
docker service ps <service ID>
```

SSH into that node, find the container ID with `docker container ls`, and check its logs with `docker container logs <container ID>`. The results of the ongoing ping should be visible.

25.2 Scaling a Service

Scale up the number of concurrent tasks that our `alpine` service is running:

```
docker service update <service name> --replicas=8
```

Now run `docker service ps <service name>` to inspect the service. Are all the containers running right away? How were they distributed across your swarm?

25.3 Cleanup

Remove all existing services, in preparation for future exercises:

```
docker service rm $(docker service ls -q)
```

25.4 Conclusion

In this example, we saw the basic syntax for defining a service based on an image, and for changing the number of replicas, or concurrent containers, running of that image. We also saw how to investigate the state of services on our swarm with `docker swarm ls` and `docker swarm ps`.

26 Load Balancing & the Routing Mesh

In this exercise, you will observe the behaviour of the built in load balancing abilities of the Ingress network.

26.1 Deploy a service

1. Start by deploying a simple service which spawns containers that echo back their hostname when `curl`'ed:

```
docker service create --name who-am-I --publish 8000:8000 --replicas 3 training/whoami:latest
```

26.2 Observe load-balancing and Scale

1. Run `curl localhost:8000` and observe the output. You should see something similar to the following:

```
$ curl localhost:8000
I'm a7e5a21e6e26
```

2. Take note of the response. In this example, our value is `a7e5a21e6e26`. The `whoami` containers uniquely identify themselves by returning their respective hostname. So each one of our `whoami` instances should have a different value.
3. Run `curl localhost:8000` again. What can you observe?
4. Notice how the value changes each time. This shows us that the routing mesh has sent our 2nd request over to a different container, since the value was different.
5. Repeat the command two more times. What can you observe? You should see one new value and then on the 4th request it should revert back to the value of the first container. In this example that value is `a7e5a21e6e26`
6. Now let's scale the number of Tasks for our `who-am-I` service. We will add 3 more Tasks to the service so that we have a total of six `whoami` containers. Then check to see if all your containers are running:

```
docker service update who-am-I --replicas=6
docker service ps who-am-I
```

7. Now run curl localhost:8000 multiple times again. Use a script like this:

```
$ for n in {1..10}; do
>   curl localhost:8000
> done;
I'm 263fc24d0789
I'm 57ca6c0c0eb1
I'm c2ee8032c828
I'm c20c1412f4ff
I'm e6a88a30481a
I'm 86e262733b1e
I'm 263fc24d0789
I'm 57ca6c0c0eb1
I'm c2ee8032c828
I'm c20c1412f4ff
```

You should be able to observe some new values. Note how the values repeat after the 6th curl command.

26.3 The Routing Mesh

Run an nginx service and expose the service port 80 on port 8080:

```
docker service create --name nginx --publish 8080:80 nginx
```

Check which node your nginx service task is scheduled on:

```
docker service ps nginx
```

Open a web browser and hit the IP address of that node at port 8080. You should see the NGINX welcome page. Try the same thing with the IP address of any other node in your cluster (using port 8080). You should still be able to see the NGINX welcome page due to the routing mesh.

26.4 Cleanup

Remove all existing services, in preparation for future exercises:

```
docker service rm $(docker service ls -q)
```

26.5 Conclusion

In these examples, you saw that requests to an exposed service will be automatically load balanced across all tasks providing that service. Furthermore, exposed services are reachable on all nodes in the swarm - whether they are running a container for that service or not.

27 Node Failure Recovery

In Swarm Mode, services created with `docker service create` are primary; if something goes wrong with the cluster, the manager leader does everything possible to restore the state of all services.

27.1 Set up a Service

Set up an `nginx` service with four replicas on your manager node:

```
manager$ docker service create --replicas 4 --name nginx nginx
```

Now watch the output of `docker service ps` on the same node:

```
manager$ watch docker service ps nginx
```

27.2 Simulate Node Failure

SSH into the one non-manager node in your swarm, and simulate a node failure by rebooting it:

```
worker$ sudo reboot now
```

Back on your manager node, watch the updates to `docker service ps`; what happens to the task running on the rebooted node?

27.3 Cleanup

Remove all existing services, in preparation for future exercises:

```
manager$ docker service rm $(docker service ls -q)
```

27.4 Conclusion

In this exercise, you saw Swarm Mode's scheduler in action - when a node is lost from the swarm, tasks are automatically rescheduled to restore the state of our services.

28 Dockercoins On Swarm

In this example, we'll go through the preparation and deployment of a sample application, our dockercoins miner, on our swarm. We'll define our app using a `docker-compose.yml` file, and deploy is as our first example of a stack.

28.1 Prepare Service Images

If you haven't done so already, follow the 'Prepare Service Images' step in the 'Starting a Compose App' exercise in this book, on your manager node. In this step, you built all the images you need for your app and pushed them to Docker Hub, so they'd be available for every node in your swarm.

28.2 Start our Services

Now that everything is prepped, we can start our stack. On the manager node:

```
docker stack deploy -c=docker-compose.yml dc
```

Check and see how your services are doing:

```
docker stack services dc
```

Notice the `REPLICAS` column in the output of above command; this shows how many of your desired replicas are running. At first, a few might show 0/1; before those tasks can start, the worker nodes will need to download the appropriate images from Docker Hub. Wait a minute or two, and try `docker stack services dc` again; once all services show 100% of their replicas are up, things are running properly and you can point your browser to port 8000 on one of the swarm nodes (does it matter which one)? You should see a graph of your dockercoin mining speed, around 3 hashes per second.

Finally, check out the details of the tasks running in your stack with `stack ps`:

```
docker stack ps dc
```

This shows the details of each running container involved in your stack - if all is well, there should be five, one for each service in the stack.

29 Scaling and Scheduling Services

29.1 Scaling up a Service

If we've written our services to be stateless, we might hope for linear performance scaling in the number of replicas of that service. For example, our `worker` service requests a random number from the `rng` service and hands it off to the `hasher` service; the faster we make those requests, the higher our throughput of dockercoins should be, as long as there are no other confounding bottlenecks. We can define the number of replicas for a service in our `docker-compose.yml` by modifying the `worker` service definition with a `deploy` key:

```
  worker:
    image: user/dockercoins_worker:1.0
    networks:
```

```
    - dockercoins
    deploy:
      replicas: 2
```

Make this modification, and update your app by running the same command you used to launch it in the first place:

```
docker stack deploy -c=docker-compose.yml dc
```

Note that running `docker stack deploy` on a stack that's already running will only apply the changes made to the stack since the last call to `stack deploy`. Once both replicas of the `worker` service are live, check the web frontend; you should see about double the number of hashes per second, as expected. Scale up even more by changing the `worker` replicas to 10. A small improvement should be visible, but certainly not an additional factor of 5. Something else is bottlenecking dockercoins.

29.2 Scheduling Services

Something other than `worker` is bottlenecking dockercoins's performance; the first place to look is in the services that `worker` directly interacts with. The `rng` and `hasher` services are exposed on host ports 8001 and 8002, so we can use `httping` to probe their latency:

```
httping -c 10 localhost:8001
httping -c 10 localhost:8002
```

`rng` is much slower to respond, suggesting that it might be the bottleneck. If this random number generator is based on an entropy collector (random voltage microfluctuations in the machine's power supply, for example), it won't be able to generate random numbers beyond a physically limited rate; we need more machines collecting more entropy in order to scale this up. This is a case where it makes sense to run exactly one copy of this service per machine, via `global` scheduling (as opposed to potentially many copies on one machine, or whatever the scheduler decides as in the default `replicated` scheduling).

Modify the definition of our `rng` service in `docker-compose.yml` to be globally scheduled:

```
  rng:
    image: user/dockercoins_rng:1.0
    networks:
    - dockercoins
    ports:
    - "8001:80"
    deploy:
      mode: global
```

Scheduling can't be changed on the fly, so we need to stop our app and restart it:

```
docker stack rm dc
docker stack deploy -c=docker-compose.yml dc
```

Check the web frontend again; the overall factor of 10 improvement (from ~3 to ~35 hashes per second) should now be visible.

29.3 Conclusion

In this exercise, you explored the performance gains a distributed application can enjoy by scaling a key service up to have more replicas, and by correctly scheduling a service that needs to be replicated across physically different nodes.

30 Updating a Service

30.1 Rolling Updates

First, let's change one of our services a bit: open

`orchestration-workshop/dockercoins/worker/worker.py` in your favorite text editor, and find the following section:

```
def work_once():
    log.debug("Doing one unit of work")
     time.sleep(0.1)
```

Change the `0.1` to a `0.01`. Save the file, exit the text editor, rebuild the image with a tag of `<Docker Hub username>/dockercoins_worker:1.1`, and push it to Docker Hub.

Next, open a new ssh connection to your manager node, and set it to watch the following:

```
watch -n1 "docker service ps dc_worker | grep -v Shutdown.*Shutdown"
```

(this last step isn't necessary to do an update, but is just for illustrative purposes to help us watch the update in action). Switch back to your original connection to your manager, and start the update:

```
docker service update dc_worker --image <user>/dockercoins_worker:1.1
```

Switch back to your new terminal and observe the output. You should notice the tasks images being updated to our new 1.1 image one at a time.

30.2 Parallel Updates

We can also set our updates to run in batches by configuring some options associated with each service. On your first connection to your manager, change the parallelism to 2 and the delay to 5 seconds on the `worker` service by editing its definition in the `docker-compose.yml`:

```
  worker:
    image: billmills/dockercoins_worker:1.0
    networks:
    - dockercoins
    deploy:
      replicas: 10
      update_config:
        parallelism: 2
        delay: 5s
```

Roll back the worker service to 1.0:

```
docker stack deploy -c=docker-compose.yml dc
```

On the second connection to manager, the `watch` should show instances being updated two at a time, with a five second pause between updates.

30.3 Shutting Down a Stack

To shut down a running stack:

```
docker stack rm <stack name>
```

Where the stack name can be found in the output of `docker stack ls`.

31 Docker Secrets

Docker offers tooling to manage secrets securely, ensuring they are never transmitted or stored unencrypted on disk. In this exercise, we will explore basic secret usage.

Note that secrets management is available only in Swarm mode; make sure you're running in Swarm mode and working on a Swarm manager node before proceeding.

31.1 Creating Secrets

Create a new secret named `my-secret` with the value `some sensitive value` by using the following command to pipe STDIN to the secret value:

```
$ echo 'my sensitive value' | docker secret create my-secret -
```

The response of this command (if successful) should be the `ID` of the newly created secret, e.g. `o6v1pp3exc8sjz4nlibk5nkto`.

Alternatively, secret values can be read from a file. In the current directory create a file called `mysql-password.txt` and add the value `1PaSsw0rd2` to it. Create a secret with this value:

```
$ docker secret create mysql-password ./mysql-password.txt
```

31.2 Creating secrets with labels

During creating of a secret we can also assign labels to it using the flags `-l` or `--label`:

```
$ echo 'abra-cadabra' | docker secret create -l env=PROD -l version=1.1 another-secret -
$ docker secret inspect another-secret
```

and you should see something like this (please note the labels)

```
[
    {
        "ID": "mkidvrm19juprj4mg6j259b0q",
        "Version": {
            "Index": 135
        },
        "CreatedAt": "2017-01-25T15:21:35.845384032Z",
        "UpdatedAt": "2017-01-25T15:21:35.845384032Z",
        "Spec": {
            "Name": "another-secret",
            "Labels": {
                "env": "PROD",
                "version": "1.1"
            }
        }
    }
]
```

31.3 Listing and inspecting secrets

Secrets are stored encrypted in the Raft log of the Swarm. Only manager nodes have access to those (encrypted) secrets. To get a list of all secrets available on the current manager node, do:

```
$ docker secret ls
```

You should see a list similar to this:

```
ID                       NAME             CREATED            UPDATED
98a432nhdgsg42yryeybivf0y  my-secret       17 hours ago      17 hours ago
mkidvrm19juprj4mg6j259b0q  another-secret    About a minute ago  About a minute ago
oqoue3lbin3mr83hd46ansnrf  mysql-password    13 minutes ago     13 minutes ago
```

If we want to see a more thorough list of all (meta-)data associated with a particular secret then we can use the `inspect` command:

```
$ docker secret inspect my-secret
```

and we should get something along the lines of:

```
[
    {
        "ID": "98a432nhdgsg42yryeybivf0y",
        "Version": {
            "Index": 15
        },
        "CreatedAt": "2017-01-24T21:52:12.221392326Z",
        "UpdatedAt": "2017-01-24T21:52:12.221392326Z",
        "Spec": {
            "Name": "my-secret"
        }
    }
]
```

31.4 Deleting Secrets

Secrets can be removed from storage using the `rm` command:

```
# create a sample secret
$ docker secret create PROD_LICENSE ./mysql-password.txt

# and delete it
$ docker secret rm PROD_LICENSE
```

31.5 Using Secrets

Secrets are assigned to Swarm services upon creation of the service. Let's create a simple service using the `alpine` image.

```
$ docker service create \
    --name demo \
    --secret my-secret \
    --secret mysql-password \
    alpine:latest ping 8.8.8.8
```

Use `docker service ps demo` to determine what node your service container is running on; ssh into that node, and connect to the container (remember to use `docker container ls` to find the container ID):

```
$ docker container exec -it <container ID> sh
```

Secrets are mounted via an in-memory filesystem at `/run/secrets`:

```
$ cd /run/secrets
$ ls
$ cat my-secret
$ exit
```

This is the *only* place secret values sit unencrypted in memory.

31.6 Updating a Secret

Secrets are never updated in-flight; secrets are added or removed, restarting the service each time. Let's create a new version of the `my-secret` secret; we'll add the `-v2` suffix just to distinguish it from the original secret, in case we need to roll back:

```
$ echo 'updated value v2' | docker secret create my-secret-v2 -
```

and update our demo service first by deleting the old secret:

```
$ docker service update --secret-rm my-secret demo
```

note how the service needs to be restarted for this operation:

```
$ docker service ps demo
ID           NAME       IMAGE        NODE  DESIRED STATE  CURRENT STATE         ERROR  PORTS
5gh2itpwf5ds  demo2.1    ubuntu:14.04  moby  Ready        Ready 9 seconds ago
x2ea5k4she9t  \_ demo2.1  ubuntu:14.04  moby  Shutdown     Running 9 seconds ago
```

Finally we assign the new value of the secret to the service, using `source` and `target` to alias the `my-secret-v2` outside the container as `my-secret` inside:

```
$ docker service update --secret-add source=my-secret-v2,target=my-secret demo
```

If we now exec into the running container we should find the new and update value

```
manager$ docker service ps demo  # find the node the container is running on, 'worker'
worker$ docker container ls .    # find the ID of the container
worker$ docker container exec -it <container ID> bash

container$ cd /run/secrets
container$ cat my-secret
updated value v2
container$ exit
```

31.7 Preparing an image for use of secrets

Containers need to consume secrets per their mounting in `/run/secrets`. In many cases, existing application logic expects secret values to appear behind environment variables; in the following, we set up such a situation as an example.

Create a new directory `image-secrets` and navigate to this folder. In this folder create a file named `app.py` and add the following content

```
import os
print '***** DOCKER Secrets ******'
print 'USERNAME: {0}'.format(os.environ['USERNAME'])

fname = os.environ['PASSWORD_FILE']
with open(fname) as f:
    content = f.readlines()
```

```
print 'PASSWORD_FILE: {0}'.format(fname)
print 'PASSWORD: {0}'.format(content[0])
```

Now add a file called `Dockerfile` with the this content

```
FROM python:2.7
RUN mkdir -p /app
WORKDIR /app
COPY . /app
CMD python ./app.py && sleep 1000
```

build the image and push it to a registry so it's available to all nodes in your swarm:

```
$ docker image build -t <username>/secrets-demo:1.0 .
$ docker image push  <username>/secrets-demo:1.0
```

create and run a service using this image:

```
$ docker service create \
    --name secrets-demo \
    --replicas=1 \
    --secret source=mysql-password,target=db_password,mode=0400 \
    -e USERNAME="jdoe" \
    -e PASSWORD_FILE="/run/secrets/db_password" \
    <username>/secrets-demo:1.0
```

Figure out which node your container is running on, head over there, connect to the container, and run `python app.py`; the `-e` flag in `service create` has set environment variables to point at your secrets, allowing your app to find them where it expects.

31.8 Conclusion

In this lab we have learned how to create, inspect and list secrets. We also have seen how we can assign secrets to services and investigated how containers actually get the secrets. We then updated the secret of a given service, and finally we created an image that is prepared to consume secrets.