Docker Fundamentals Exercises

These exercises were validated on Centos 7.3 and Docker 17.06-EE. Instructors, please ensure student lab environments match this stack.

Windows Users: Please note that in all exercises we will use Unix style paths using forward slashes ('/') instead of backslashes ('\'). On Windows you can work directly with such paths by either using a **Bash** terminal or a **Powershell** terminal. Powershell can work with both Windows and Unix style paths.

Contents

1 Running & Inspecting a Container

1.1 Running Containers

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

   ```
   $ docker container run centos:7 echo "hello world"
   ```

   The `centos:7` 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.

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

   ```
   $ docker container run  centos:7 ps -ef
   ```

3. `ps -ef` was PID 1 inside the container in the last step; try doing `ps -ef` at the host prompt and see what process is PID 1 here.

1.2 Listing Containers

1. 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.

2. 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; this process is a member of the host's PID tree like any other, but is 'containerized' via tools like kernel namespaces, making this process and its children behave as if it was the root of a PID tree, with its own user ID spectrum, filesystem, and network stack. Furthermore, this process defines the state of the container; if the process exits, the container stops.

2 Interactive Containers

2.1 Writing to Containers

1. Create a container using the `centos:7` image, and connect to its bash shell in interactive mode using the `-i` flag (also the `-t` flag, to request a TTY connection):

   ```
   $ docker container run -it centos:7 bash
   ```

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

   ```
   $ ls -l
   $ echo 'Hello there...' > test.txt
   $ ls -l
   ```

3. Exit the connection to the container:

   ```
   $ exit
   ```

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


```
$ docker container run -it centos:7 bash
```

5. Try finding your `test.txt` file inside this new container; it is nowhere to be found. Exit this container for now in the same way you did above.

2.2 Reconnecting to Containers

1. We'd like to recover the information written to our container in the first example, but starting a new container 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
   ```

2. We can restart a container via the container ID listed in the first column. Use the container ID for the first `centos:7` container you created with `bash` as its command:

   ```
   $ docker container start <container ID>
   ```

3. Run `ps -ef` inside the container you just restarted using Docker's `exec` command (this can be used to execute *any* command in a running Docker container):

   ```
   $ docker container exec <container ID> ps -ef
   ```

   What process is PID 1 inside the container? What is the PID of that process on the host machine?

4. Run a command in your running container with `docker container exec`:

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

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

2.3 Listing Container Options

1. 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`.

2. List only the container ID using the `-q` flag:

   ```
   $ docker container ls -a -q
   ```

3. List the last container to have been created using the `-l` flag:

   ```
   $ docker container ls -l
   ```

4. 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"
   ```

   The outputs of these commands will list the containers that have exited successfully or not.

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, and exist only in that particular container's R/W layer, of which there is one per container. 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

1. 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 centos:7 ping 127.0.0.1 -c 10
   ```

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

   ```
   $ docker container run -d centos:7 ping 127.0.0.1
   ```

3. 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

1. 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>
   ```

2. 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.

3. Try running the same thing in detached interactive mode:

   ```
   $ docker container run -d -it centos:7 ping 127.0.0.1
   ```

4. 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 Using Logging Options

1. 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>
   ```

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

   ```
   $ docker container logs -f <container ID>
   ```

   (`CTRL+C` to break out of following mode).

3. 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

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

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

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

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

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

```
$ docker container start -a <container ID>
```

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

5. Finally, stop the container with `docker container kill`:

```
$ docker container kill <container ID>
```

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

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

```
$ docker container start <container ID>
$ docker container inspect <container ID>
```

2. 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
```

The output should look similar to this:

```
    "SecondaryIPAddresses": null,
    "IPAddress": "<Your IP Address>"
```

3. 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:

```
    "Cmd": [
```

4. 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>
```

5. 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.

6. 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

1. Start three containers in background mode, then stop the first one.
2. List only exited containers using the `--filter` flag we learned earlier, and the option `status=exited`.
3. 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 ...
```

4. 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>
```

5. 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 Making an Account on Docker's Hosted Registry

1. If you don't have one already, head over to http://store.docker.com and make an account. This account is synchronized across three services:

- **Docker Store**, for browsing official content
- **Docker Hub**, for sharing community-generated content
- **Docker Cloud**, for creating simple automated, containerized pipelines

For the rest of this workshop, `<Docker ID>` refers to the username you chose for this account.

5.2 Modifying a Container

1. Start a bash terminal in a CentOS container:

```
$ docker container run -it centos:7 bash
```

2. 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:

```
$ yum update
$ yum install -y wget vim
$ exit
```


3. 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.

5.3 Capturing Container State as an Image

1. 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> <Docker ID>/myapp:1.0
```

   Notice the image name format: `<Docker ID>/<image name>[:optional tag]`. Images *must* be named according to this pattern to push them to Docker's hosted registry.

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

```
$ docker image ls
```

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

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

4. Create a file in your container and commit that as a new image. Use the same image name but tag it as 1.1.
5. 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?

5.4 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

1. Create a folder called `myimage`, and a text file called `Dockerfile` within that folder. In the `Dockerfile`, include the following instructions:

```
FROM centos:7

RUN yum update -y
RUN yum install -y wget
```

2. Build your image with the `build` command:

```
$ docker image build -t <Docker ID>/myimage .
```

3. Verify that your new image exists with `docker image ls`, then use it to run a container and `wget` something from within that container.
4. It's also possible to pipe a Dockerfile in from STDIN; try rebuilding your image with the following:


```
$ cat Dockerfile | docker build -t <Docker ID>/myimage -f - .
```

(This is useful when reading a Dockerfile from a remote location with `curl`, for example).

6.2 Using the Build Cache

In the previous step, the second time you built your image should have completed immediately, with each step save the first reporting `using cache`. Cached build steps will be used until a change in the Dockerfile is found by the builder.

1. Open your Dockerfile and add another `RUN` step at the end to install `vim`.
2. Build the image again as above; which steps is the cache used for?
3. Build the image again; which steps use the cache this time?
4. Swap the order of the two `RUN` commands for installing `wget` and `vim` in the Dockerfile, and build one last time. Which steps are cached this time?

6.3 Using the `history` Command

1. 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 `yum update` command.
2. Replace the two `RUN` commands that installed `wget` and `vim` with a single command:

   ```
   ...
   RUN yum install -y wget vim
   ```

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

6.4 Conclusion

In this exercise, we've seen how to write a basic Dockerfile using `FROM` and `RUN` commands, some basics of how image caching works, and seen the `docker image history` command. Using the build cache effectively is crucial for images that involve lengthy compile or download steps; in general, moving commands that change frequently as late as possible in the Dockerfile will minimize build times. We'll see some more specific advice on this later in this lesson.

7 Creating Images with Dockerfiles (2/2)

7.1 Setting Default Commands

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

   ```
   ...
   CMD ["ping", "127.0.0.1", "-c", "5"]
   ```

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

   ```
   $ docker image build -t <Docker ID>/myimage:1.0 .
   ```


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

```
$ docker container run <Docker ID>/myimage:1.0
```

4. 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 <Docker ID>/myimage:1.0 echo "hello world"
```

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

5. Replace the CMD instruction in your Dockerfile with an ENTRYPOINT:

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

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

```
$ docker image build -t <Docker ID>/myimage:1.0 .
$ docker container run <Docker ID>/myimage:1.0
```

What went wrong?

7. Try running with an argument after the image name:

```
$ docker container run <Docker ID>/myimage:1.0 127.0.0.1
```

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

7.2 Combining Default Commands and Options

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

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

2. 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"]
```

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

```
$ docker container run <Docker ID>/myimage:1.0 <some public IP>
```

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

7.3 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 Multi-Stage Builds

In this exercise, you'll work with a multi-stage Dockerfile to build two versions of your image: a development version with lots of tooling in place, and a production-ready version that's as lightweight as possible.

8.1 Defining a multi-stage build

1. Make a fresh folder `multi-stage` to do this exercise in, and `cd` into it.
2. Add a file `hello.c` to the `multi-stage` folder containing **Hello World** in C:

```
#include <stdio.h>

int main (void)
{
    printf ("Hello, world!\n");
    return 0;
}
```

3. Try compiling and running this right on the host OS:

```
$ sudo yum install gcc
$ gcc -Wall hello.c -o hello
$ ./hello
```

4. Now let's Dockerize our hello world application. Add a `Dockerfile` to the `multi-stage` folder with this content:

```
FROM alpine:3.5
RUN apk update && \
    apk add --update alpine-sdk
RUN mkdir /app
WORKDIR /app
COPY hello.c /app
RUN mkdir bin
RUN gcc -Wall hello.c -o bin/hello
CMD /app/bin/hello
```

5. Build the image and observe its (massive) size:

```
$ docker image build -t my-app-large .
$ docker image ls | grep my-app-large
```

6. Update your Dockerfile to use an `AS` clause on the first line, and add a second stanza describing a second build stage:

```
FROM alpine:3.5 AS build
RUN apk update && \
    apk add --update alpine-sdk
RUN mkdir /app
WORKDIR /app
COPY hello.c /app
RUN mkdir bin
RUN gcc -Wall hello.c -o bin/hello

FROM alpine:3.5
COPY --from=build /app/bin/hello /app/hello
CMD /app/hello
```

7. Build the image again and compare the size with the previous version:

```
$ docker image build -t my-app-small .
$ docker image ls | grep 'my-app-'
```

   As expected, the size of the lean build is much smaller than the large one since it does not contain all the Alpine SDK.

8. Finally, make sure the app actually works:


```
$ docker container run --rm my-app-small
```

8.2 Building Intermediate Images

In the previous step, we took our compiled executable from the first build stage, but that image never survived the build process; only the final `FROM` statement generated an image. In this step, we'll see how to persist whichever build stage we like.

1. Build an image from the `build` stage in your Dockerfile using the `--target` flag:

   ```
   $ docker image build -t my-build-stage --target build .
   ```

2. Run a container from this image and make sure it yields the expected result:

   ```
   $ docker container run -it --rm my-build-stage /app/bin/hello
   ```

3. List your images again to see the size of `my-build-stage` compared to the small version of the app.

8.3 Conclusion

In this exercise, you created a Dockerfile defining multiple build stages. Being able to take artifacts like compiled binaries from one image and insert them into another allows you to create very lightweight images that do not include developer tools or other unnecessary components in your production-ready images. This will result in containers that start faster, and are less vulnerable to attack.

9 Managing Images

9.1 Tagging and Listing Images

1. Download the `centos:7` image from Docker Store:

   ```
   $ docker image pull centos:7
   ```

2. Make a new tag of this image:

   ```
   $ docker image tag centos:7 my-centos:dev
   ```

3. List your images:

   ```
   $ docker image ls
   ```

4. You should have `centos:7` and `my-centos: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 Docker Hub

1. 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.

2. Push your image to Docker Hub:

   ```
   $ docker image push my-centos:dev
   ```

   You should get an `authentication required` error.

3. 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 Store; all images you want to share on Docker Store must be named like `<Docker ID>/<repo name>[:<optional tag>]`.

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

```
$ docker image tag my-centos:dev <Docker ID>/my-centos:dev
$ docker image push <Docker ID>/my-centos:dev
```

5. Search Docker Store for your new `<Docker ID>/my-centos` repo, and confirm that you can see the `:dev` tag therein.

6. Next, write a Dockerfile that uses `<Docker ID>/my-centos: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 <Docker ID>/my-centos:1.0 .
```

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

8. 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-centos:dev
```

Only the tag gets deleted, not the actual image. The image is still referenced by another tag.

9.3 Using Private Images

1. Explore the UI on your Docker Cloud repo `<Docker ID>/my-centos`, and find the toggle to make that repo private.

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

```
$ docker image pull <partners Docker store name>/my-centos:1.0
```

Their private repo should be invisible to you at this time.

3. Add each other as Collaborators to your `my-centos` 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 Cloud.

10 Cleaning up Docker Resources

Once we start to use Docker heavily, we want to understand what resouces the Docker engine is using. We also want tools to free up resources. For all this we can use the `docker system` commands.

1. 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.

2. Reclaim all reclaimable space by using the following command:

```
$ docker system prune
```

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

3. Create a couple of containers with labels (these will exit immediately; why?):

```
$ docker container run --label apple --name fuji -d alpine
$ docker container run --label orange --name clementine -d alpine
```

4. Delete only those stopped containers bearing the `apple` label:

```
$ docker container ls -a
$ docker container prune --filter 'label=apple'
$ docker container ls -a
```

   Only the container named `clementine` should remain after the targeted prune.

5. Finally, prune containers launched before a given timestamp using the `until` filter; start by getting the current RFC 3339 time (https://tools.ietf.org/html/rfc3339 - note Docker *requires* the otherwise optional T separating date and time), then creating a new container:

```
$ TIMESTAMP=$(date --rfc-3339=seconds | sed 's/ /T/')
$ docker container run --label tomato --name beefsteak -d alpine
```

   And use the timestamp returned in a prune:

```
$ docker container prune -f --filter "until=$TIMESTAMP"
$ docker container ls -a
```

   Note the `-f` flag, to suppress the confirmation step. `label` and `until` filters for pruning are also available for networks and images, while data volumes can only be selectively pruned by `label`; finally, images can also be pruned by the boolean `dangling` key, indicating if the image is untagged.

11 Inspection Commands

11.1 Inspecting System Information

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

```
$ docker system info
```

2. From the output of the last command, 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?

11.2 Monitoring System Events

1. 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.

2. 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 30eb63 ...
2017-01-25T16:57:48.556718161-06:00 container attach 30eb63 ...
2017-01-25T16:57:48.698190608-06:00 network connect de1b2b ...
2017-01-25T16:57:49.062631155-06:00 container start 30eb63 ...
2017-01-25T16:57:49.065552570-06:00 container resize 30eb63 ...
2017-01-25T16:57:49.164526268-06:00 container die 30eb63 ...
2017-01-25T16:57:49.613422740-06:00 network disconnect de1b2b ...
2017-01-25T16:57:49.815845051-06:00 container destroy 30eb63 ...
```

3. 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.

4. 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 system 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 exercise 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 Creating and Mounting Volumes

12.1 Creating a Volume

1. Create a volume called `test1`:

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

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

3. Execute a new `centos` 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 centos:7 bash
```

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

```
$ cd /www/website
```

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

```
$ echo 'hello there' > test.txt
```

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

12.2 Creating Volumes with Containers

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

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

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

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

3. Verify that the `/www/website` folder exists. Are there any files inside it? Exit this container.
4. Run `docker container ls` to ensure that your first container is still running in preparation for the next step.

In this section, 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; volume data is outside the layered filesystem in this sense. Note however that the mountpoint `/www/website` *was* captured in image creation; the mountpoint directory is created in the container layer itself, while the contents of the volume remain external to it.

12.3 Finding the Host Mountpoint

1. Run `docker volume inspect` on the `test1` volume to find out where it is mounted on the host machine (see the 'Mountpoint' field):

```
$ docker volume inspect test1
```

2. Elevate your user privileges to root:

```
$ sudo su
```

3. Change directory into the volume path found in step 1:

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

4. Run `ls` and verify you can see the `test.txt` file you created inside your original container.
5. Create another file called `test2.text` inside this directory:

```
$ touch test2.txt
```

6. Exit the superuser account:

```
$ exit
```

7. Use `docker container exec` to log back into the shell of your `centos` container that is still running:

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


8. Change directory into the `/www/website` folder, and verify that you can see both the `test.txt` and `test2.txt` files. Files written to the volume on the host listed by `docker volume inspect` appear in the filesystem of every container that mounts it.

12.4 Deleting Volumes

1. After exiting your container and returning to the host's bash prompt, attempt to delete the `test1` volume:

   ```
   $ docker volume rm test1
   ```

   Deletion fails since the volume is still mounted to a container.

2. Delete the remaining container without using any options:

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

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

4. 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
   ```

5. Delete the `test1` volume:

   ```
   $ docker volume rm test1
   ```

   Deletion succeeds now that no containers are using the volume.

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

12.5 Conclusion

Volumes are the Docker-preferred tool for persisting data beyond the lifetime of a container. In this exercise, we saw how to create and destroy volumes; how to mount volumes when running a container; how to find their locations on the host (under `/var/lib/docker/volumes`) and in the container using `docker volume inspect` and `docker container inspect`; and how they interact with the container's union file system.

13 Volumes Usecase: Recording Logs

13.1 Setting up an app with logging

1. Create a volume called `nginx_logs`:

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

2. Create a folder called `public_html` inside your home directory.

3. Inside your `public_html` folder, create a file called `index.html` and write some lines of text on the file.

4. Run the custom `training/nginx:17.06` image 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 \
           training/nginx:17.06
   ```

5. 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.

6. Verify you can see the contents of your `index.html` file from your `public_html` folder on your browser.
7. Get terminal access to your container:

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

8. Put some additonal text into `/usr/share/nginx/html/index.html`, and then exit the terminal.
9. Verify you can see the contents of your updated `index.html` file from your `public_html` folder on your browser.
10. Update the `index.html` file inside your `public_html` folder again and then refresh your browser. Verify that the updated text appears.

13.2 Inspecting Logs on the Host

1. Get terminal access to your container again:

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

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

```
container$ cd /var/log/nginx
```

3. Check that you can see the `access.log` and `error.log` files.
4. 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.
5. 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`.
6. 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
```

7. 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.

13.3 Sharing Volumes

1. Run `docker container ls` and make sure that your `nginx_server` container from the last step is still running; if not, restart it.
2. Run a new `centos` 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 \
        centos:7 bash
```

3. In your new container's terminal, change directory to `/data/mylogs`
4. Confirm that you can see the `access.log` and `error.log` files.
5. 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.

13.4 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.

We also used the `:ro` flag to mount a volume in read-only mode, so the corresponding container can't damage any data in the volume. This is also an important security best practice, since sharing volumes between containers breaks isolation between containers; by allowing read-only access, we prevent the 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.

14 Docker Plugins

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

14.1 Installing a Plugin

1. Plugins can be hosted on Docker Store or any other (private) repository. Let's start with Docker Store. Browse to https://store.docker.com and enter `vieux/sshfs` in the search box. The result should show you the plugin that we are going to work with.
2. Install the plugin into our Docker Engine:

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

   The system should ask us for permission to use privileges. In the case of the `sshfs` plugin there are 4 privileges. Answer with `y`.
3. 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
   ```

14.2 Enabling and Disabling a Plugin

1. 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.
2. 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`.

14.3 Inspecting a Plugin

1. 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 examine 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.

2. 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.

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

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

14.4 Removing a Plugin

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

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

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

14.5 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.

1. To use the plugin we create a Docker volume:

```
$ 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.

2. 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
```

3. Inside the container navigate to the `/data` folder and create a new file:

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

4. If you created the volume from step 1 using the details of your own remote server, SSH into that remote server and check that the file created in step 3 exists. If you used a common remote server setup by the instructor, the instructor will navigate to the appropriate folder on that server and show the presence of the created file.

14.6 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.

15 Introduction to Container Networking

15.1 The Default Bridge Network

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

   ```
   $ sudo yum install bridge-utils
   ```

2. Ask for information about the default Docker linux bridge, `Docker0`:

   ```
   $ brctl show docker0
   ```

3. Start some named containers and check again:

   ```
   $ docker container run --name=u1 -dt centos:7
   $ docker container run --name=u2 -dt centos:7
   $ brctl show docker0
   ```

   You should see two new 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.

4. 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`.

5. Connect to container `u2` of your containers using `docker container exec -it u2 /bin/bash`.

6. From inside `u2`, 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.

7. Still inside container `u2`, install `iproute`

   ```
   yum install -y iproute
   ```

8. 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).

9. 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.

15.2 Defining Additional 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.

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

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

2. 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.

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

   ```
   $ docker container run --name=u3 --network=my_bridge -dt centos:7
   ```


4. 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.

5. Launch another container, this time interactively:

   ```
   $ docker container run --name=u4 --network=my_bridge -it centos:7
   ```

6. 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.
7. 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.
8. Exit container `u4` by pressing `CTRL+P,Q`. This will ensure that the container remains running.

15.3 Communicating Between Containers

1. 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
   ```

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

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

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

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

4. Note `u1` still can't reach `u3` and `u4`:

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

15.4 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*. This should be leveraged as much as possible to harden your applications; if two containers don't need to talk to each other, put them on separate networks.

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.

16 Container Port Mapping

16.1 Port Mapping at Runtime

1. 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.

2. Now run an nginx container and map port 80 on the container to port 5000 on your host using the `-p` flag:

   ```
   $ docker container run -d -p 5000:80 nginx
   ```

   Note that the syntax is: `-p [host-port]:[container-port]`.

3. Verify the port mappings with the `docker container port` command

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

4. Visit your nginx landing page at `<host ip>:5000`, e.g. using `curl localhost:5000`.

16.2 Exposing Ports from the Dockerfile

1. 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
   ```

2. Build your image as `my_nginx`:

   `bash $ docker image build -t my_nginx .`

3. Use the `-P` flag when running to map all ports mentioned in the `EXPOSE` directive:

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

4. 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.

16.3 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.

17 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.

17.1 Preparing Service Images

1. Download the Dockercoins app from github:

```
$ git clone -b 17.06 https://github.com/docker-training/orchestration-workshop.git
```

   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, which we will build and push to Docker Store for later use (if you don't have a Docker Store account, make a free one first at https://store.docker.com).

2. Log into your Docker Store account from the command line:

```
$ docker login
```

3. Build and push the images corresponding to the `rng`, `hasher`, `worker` and `webui` services. For `hasher`, this looks like (from the `orchestration-workshop/dockercoins` folder you just cloned from GitHub):

```
$ docker image build -t <Docker ID>/dockercoins_hasher:1.0 hasher
$ docker image push <Docker ID>/dockercoins_hasher:1.0
```

   Repeat this three more times, for `rng`, `worker`, and `webui`.

4. Look in `docker-compose.yml`, and change all the `image` values to have your Docker 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 Store.

17.2 Starting the App

1. 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
```

2. 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
```

3. Check out which containers are running thanks to Compose:

```
$ docker-compose ps
```

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

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

17.3 Viewing Logs

1. See logs from a Compose-managed app via:

```
$ docker-compose logs
```

2. 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.

17.4 Conclusion

In this exercise, you saw how to start a pre-defined Compose app, and how to inspect its logs. Application logic was defined in each of the five images we used to create containers for the app, but the manner in which those containers were created was defined in the `docker-compose.yml` file; every aspect of how many containers we want for each service, what networks to attach those containers to and what other parameters are desired, is captured in this manifest. Finally, the different elements of Dockercoins communicated with each other via service name; the Docker daemon's internal DNS was able to resolve traffic destined for a service, into the IP or MAC address of the corresponding container.

18 Scaling a Compose App

18.1 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.

1. 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
   ```

2. 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.

18.2 Investigating Bottlenecks

1. 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
   ```

2. 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.

3. 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. First we need to install `httping`:

   ```
   $ wget http://dl.fedoraproject.org/pub/epel/7/x86_64/h/httping-2.5-1.el7.x86_64.rpm
   $ sudo yum localinstall -y httping-2.5-1.el7.x86_64.rpm
   ```

4. Now we can use this tool to probe the 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.

5. For now, shut your app down:

```
$ docker-compose down
```

18.3 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.

19 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.

19.1 Start Swarm Mode

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

   ```
   $ docker swarm init
   ```

2. Confirm that Swarm Mode is active by inspecting the output of:

   ```
   $ docker system info
   ```

3. See all nodes currently in your swarm by doing:

   ```
   $ docker node ls
   ```

   A single node is reported in the cluster.

4. Change the certificate rotation period from the default of 90 days to one week, and rotate the certificate now:

   ```
   $ docker swarm ca --rotate --cert-expiry 168h
   ```

   Note that the `docker swarm ca [options]` command *must* receive the `--rotate` flag, or all other flags will be ignored.

5. Display UDP and TCP activity on your manager:

   ```
   $ sudo netstat -plunt
   ```

   You should see (at least) TCP+UDP 7946, UDP 4789, and TCP 2377. What are each of these ports for?

19.2 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.

1. On your manager node, get the swarm 'join token' you'll use to add worker nodes to your swarm:

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

2. SSH into a second node, and paste the result of the last step there. This new node will have joined the swarm as a worker.

3. Inspect the network on your worker node with `sudo netstat -plunt` like you did for the manager node. Are the same ports open? Why or why not?

4. 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.

5. 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.

19.3 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.

1. Promote two of your workers to manager status by executing, on the current manager node:

   ```
   $ docker node promote node-1 node-2
   ```

   where `node-1` and `node-2` 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).

2. 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.

19.4 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`.

20 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.

20.1 Creating an Overlay Network and Service

1. Create a multi-host overlay network to connect your service to:

`bash    $ docker network create --driver overlay my_overlay`

2. Create a service featuring an `alpine` container pinging Google resolvers, plugged into your overlay network:

   ```
   $ docker service create --name pinger --network my_overlay 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`).

3. Get some information about the currently running services:

   ```
   $ docker service ls
   ```

4. Check which node the container was created on:

   ```
   $ docker service ps pinger
   ```

5. SSH into the node you found in the last step, 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.

6. Inspect the `my_overlay` network on the current node:

```
$ docker network inspect my_overlay
```

You should be able to see the container connected to this network, and a list of swarm nodes connected to this network under the `Peers` key. Also notice the correspondence between the container IPs and the subnet assigned to the network under the `IPAM` key - this is the subnet from which container IPs on this network are drawn.

7. Connect to a node that does not have a container running from your `pinger` service, and list the networks on this machine:

```
$ docker network ls
```

`my_overlay` does not appear. Overlay networks only create a VXLAN tunnel endpoint on nodes that actually have a container plugged into them.

20.2 Scaling a Service

1. Back on a manager node, scale up the number of concurrent tasks that our `alpine` service is running:

```
$ docker service update pinger --replicas=8
```

2. Now run `docker service ps pinger` to inspect the service. Are all the containers running right away? How were they distributed across your swarm?
3. Use `docker network inspect my_overlay` again on a node that has a `pinger` container running. More nodes appear connected to this network under the `Peers` key.

20.3 Inspecting Service Logs

1. In a previous step, you looked at the container logs for an individual task in your service; manager nodes can assemble all logs for all tasks of a given service by doing:

```
$ docker service logs pinger
```

2. If instead you'd like to see the logs of a single task, on a manager node run `docker service ps pinger`, choose any task ID, and run `docker service logs <task ID>`. The logs of the individual task are returned; compare this to what you did above to fetch the same information with `docker container logs`.

20.4 Scheduling Topology-Aware Services

By default, the Swarm scheduler will spread containers across nodes based on availability, but in practice it is wise to consider datacenter segmentation; spreading tasks across datacenters or availability zones keeps the service available even when one such segement goes down.

1. Add a label `datacenter` with value `east` to two nodes of your swarm:

```
$ docker node update --label-add datacenter=east node-0
$ docker node update --label-add datacenter=east node-1
```

2. Add a label `datacenter` with value `west` to the other two nodes:

```
$ docker node update --label-add datacenter=west node-2
$ docker node update --label-add datacenter=west node-3
```

3. Create a service using the `--placement-pref` flag to spread across node labels:

```
$ docker service create --name my_proxy --replicas=2 --publish 8000:80 \
    --placement-pref spread=node.labels.datacenter \
    nginx
```


There should be `nginx` containers present on nodes with every possible value of the `node.labels.datacenter` label.

4. Use `docker service ps...` as above to check that replicas got spread across the datacenter labels.

20.5 Using a few flags

1. If a container doesn't need to write to its filesystem, it should *always* be run in read-only mode, for security purposes. Update your service to use read-only containers:

```
$ docker service update pinger --read-only
```

Try connecting to the container and creating a file to convince yourself this worked as expected.

2. When starting a service based on an image with a pre-defined `ENTRYPOINT`, issuing the command to run after the image name as above won't work. Override any existing `ENTRYPOINT` with the `--entrypoint` flag:

```
$ docker service create --entrypoint "ping 8.8.8.8" alpine
```

3. Services by default start in the background, but can be started in synchronous mode instead with the `--detach=false` flag. See this in action:

```
docker service create --detach=false \
  --replicas 5 \
  busybox top
```

20.6 Cleanup

1. Remove all existing services, in preparation for future exercises:

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

20.7 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 service ls` and `docker service ps`, how to fetch logs from a manager node for one or all the tasks in a service using `docker service logs`, how to ensure containers are scheduled across availability zones, and some particularly useful flags for use with service creation.

21 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.

21.1 Set up a Service

1. Set up an `nginx` service with four replicas on one of your manager nodes:

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

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

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


21.2 Simulate Node Failure

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

```
worker$ sudo reboot now
```

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

21.3 Cleanup

1. Remove all existing services, in preparation for future exercises:

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

21.4 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.

22 Load Balancing & the Routing Mesh

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

22.1 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
```

22.2 Observe load-balancing and Scale

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

```
$ curl -4 localhost:8000
I'm a7e5a21e6e26
```

   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.

2. Run `curl localhost:8000` again. What can you observe? 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.

3. 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`

4. Scale the number of Tasks for our `who-am-I` service to 6:

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

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


```
$ for n in {1..10}; do curl localhost:8000 -4; 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.

22.3 The Routing Mesh

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

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

2. Check which node your nginx service task is scheduled on:

   ```
   $ docker service ps nginx
   ```

3. 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.

22.4 Cleanup

1. Remove all existing services, in preparation for future exercises:

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

22.5 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.

23 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.

23.1 Prepare Service Images

1. 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 Store, so they'd be available for every node in your swarm.

23.2 Start our Services

1. Now that everything is prepped, we can start our stack. On the manager node and from `orchestration-workshop/dockercoin`

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

2. 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 Store.

3. 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.

4. 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.

24 Scaling and Scheduling Services

24.1 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.

1. Modify the `worker` service definition in `docker-compose.yml` to set the number of replicas to create using the `deploy` key:

   ```
   worker:
     image: user/dockercoins_worker:1.0
     networks:
     - dockercoins
     deploy:
        replicas: 2
   ```

2. 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
   ```

3. 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.

4. 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.

24.2 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.

1. The rng and hasher services are exposed on host ports 8001 and 8002, so we can use httping to probe their latency:

   If you have not already done so in a previous exercise, make sure you have httping installed:

   ```
   $ wget http://dl.fedoraproject.org/pub/epel/7/x86_64/h/httping-2.5-1.el7.x86_64.rpm
   $ sudo yum localinstall -y httping-2.5-1.el7.x86_64.rpm
   ```

   and then:

   ```
   $ 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).

2. 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
   ```

3. 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
   ```

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

24.3 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.

25 Updating a Service

25.1 Creating Rolling Updates

1. 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.

2. Rebuild the worker image with a tag of <Docker ID>/dockercoins_worker:1.1, and push it to Docker Store.

3. 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).

4. Switch back to your original connection to your manager, and start the update:

   ```
   $ docker service update dc_worker --image <Docker ID>/dockercoins_worker:1.1
   ```

5. 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.

25.2 Parallelizing Updates

1. 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: <Docker ID>/dockercoins_worker:1.0
       networks:
       - dockercoins
       deploy:
         replicas: 10
         update_config:
           parallelism: 2
           delay: 5s
   ```

2. Roll back the worker service to 1.0:

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

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

25.3 Configuring Rollback Contingencies

In the event of an application or container failure on deployment, we'd like to automatically roll the update back to the previous version.

1. Update the `worker` service with some parameters to define rollback:

   ```
   docker service update \
       --update-failure-action=rollback \
       --rollback-parallelism=2 \
       --rollback-monitor=20s \
       --rollback-max-failure-ratio=0.2 \
       dc_worker
   ```

   These parameters will trigger a rollback, two containers at a time, if more than 20% of services tasks fail in the first 20 seconds after an update. If any of the pre-rollback containers didn't fail, an `order: start-first` key indicates that the rollback containers should start *before* the remaining containers from the failed update are killed off.

2. Make a broken version of the `worker` service to trigger a rollback with; try removing all the `import` commands at the top of `worker.py`, for example. Then rebuild the worker image with a new tag, and attempt to update your service:


```
$ docker image build -t <Docker ID>/dockercoins_worker:bugged .
$ docker image push <Docker ID>/dockercoins_worker:bugged
$ docker service update dc_worker --image <Docker ID>/dockercoins_worker:bugged
```

3. The connection to your manager running `watch` should show the `:bugged` tag getting deployed, failing, and rolling back to `:1.0` automatically.

25.4 Shutting Down a Stack

1. 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`.

26 Docker Secrets

Docker Swarm mode 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.

26.1 Creating Secrets

1. 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 -
   ```

   Note this won't work if your machine isn't in Swarm mode!

2. 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
   ```

26.2 Managing Secrets

The Docker CLI provides API objects for managing secrets similar to all other Docker assets:

1. List your current secrets:

   ```
   $ docker secret ls
   ```

2. Print secret metadata:

   ```
   $ docker secret inspect <secret name>
   ```

3. Delete a secret (don't delete `my-secret` or `mysql-secret`, since we'll need them in the next step):

   ```
   $ docker secret rm <secret name>
   ```

26.3 Using Secrets

Secrets are assigned to Swarm services upon creation of the service, and provisioned to containers for that service as they spin up.

1. Create a service authorized to use the secrets `my-secret` and `mysql-password` secret.


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

2. 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
```

3. Inspect the secrets in this container where they are mounted by default, at `/run/secrets`:

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

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

26.4 Updating a Secret

Secrets are never updated in a running container; secrets are added or removed, restarting the service each time.

1. Create a new version of the `my-secret` secret; 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 -
```

2. Update our `demo` service first by deleting the old secret:

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

3. 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, and also to mount the secret somewhere other than `/run/secrets`:

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

4. `exec` into the running container and demonstrate that the value of the `my-secret` secret has changed.

26.5 Preparing an image for use of secrets

Containers need to consume secrets from their mountpoint, either `/run/secrets` by default, or wherever they were placed as per the example above. 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.

1. 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; this is a Python script that consumes a password from a file with a path specified by the environment variable `PASSWORD_FILE`:

```
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])
```

2. Create a file called `Dockerfile` with the following content:

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

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

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

4. Create and run a service using this image, and use the `-e` flag to create environment variables that point to your secrets:

```
$ 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" \
    <Docker ID>/secrets-demo:1.0
```

5. 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.

26.6 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.

