Docker Fundamentals for Windows Exercises

Please note that examples of expected output have in some places been truncated or abbreviated to better fit this document. These examples are rough guides only; if you see more information in your terminal than in the text, that is expected.

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.

Be aware that copy-pasting of commands or code snippets from this PDF may apply changes to some characters e.g. quotes, tabs, which may leads to errors. Please consider typing suggested commands and code snippets in case you encounter any issues.

Introduction to Strigo: Strigo is our lab platform, which your instructor will provide you with access credentials for. Log in right away and get familiar with the interface by following the Strigo primer: https://bit.ly/2SIkQL6.

Contents

Exercises

1 Running & Inspecting Containers

By the end of this exercise, you should be able to:

- Start a container
- List containers in a couple of different ways
- Query the `docker` command line help
- Remove containers

1. Create and start a new nanoserver container running `ping` to 8.8.8.8:

   ```
   PS: node-0 Administrator> docker container run `
   ```



```
    mcr.microsoft.com/windows/nanoserver:10.0.17763.737 ping 8.8.8.8 -n 3

Pinging 8.8.8.8 with 32 bytes of data:
Reply from 8.8.8.8: bytes=32 time=2ms TTL=113
Reply from 8.8.8.8: bytes=32 time<1ms TTL=113
Reply from 8.8.8.8: bytes=32 time<1ms TTL=113

Ping statistics for 8.8.8.8:
    Packets: Sent = 3, Received = 3, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 2ms, Average = 0ms
```

2. This first container sent its STDOUT to your terminal; create a second container, this time in *detatched mode*, and let it run indefinitely:

```
PS: node-0 Administrator> docker container run --detach `
    mcr.microsoft.com/windows/nanoserver:10.0.17763.737 ping 8.8.4.4 -t

4bc814e2257b2d1046c10e563b553279b7e11e8bf6309eff9047d2dfb086900f
```

   Instead of seeing the executed command (`ping 8.8.4.4 -t`), Docker engine displays a long hexidecimal number, which is the full *container ID* of your new container. The container is running detached, which means the container is running as a background process, rather than printing its STDOUT to your terminal.

3. List the running Docker containers using the `docker container ls` container command. You will see only one container running.

```
PS: node-0 Administrator> docker container ls

CONTAINER ID  IMAGE                          COMMAND             ...  STATUS
4bc814e2257b  nanoserver:10.0.17763.737  "ping 8.8.4.4 -t"  ...  Up 53 seconds
```

4. Now you know that the `docker container ls` command only shows running containers. You can show all containers that exist (running or stopped) by using `docker container ls --all`. Your container ID and name will vary. Note that you will see two containers: a stopped container and a running container.

```
PS: node-0 Administrator> docker container ls --all

CONTAINER ID  IMAGE       COMMAND              STATUS
4bc814e2257b  nanoserver  "ping 8.8.4.4 -t"    Up About a minute
7aea600a8c76  nanoserver  "ping 8.8.8.8 -n 3"  Exited (0) 3 minutes
```

   Where did those names come from? The table above has been truncated for readability, but in your output you should also see a `NAME` column on the right. All containers have names, which in most Docker CLI commands can be substituted for the container ID as we'll see in later exercises. By default, containers get a randomly generated name of the form `<adjective>_<scientist / technologist>`, but you can choose a name explicitly with the `--name` flag in `docker container run`.

5. Start up another detached container, this time giving it a name "opendnsping".

```
PS: node-0 Administrator> docker container run --detach --name opendnsping `
    mcr.microsoft.com/windows/nanoserver:10.0.17763.737 ping 208.67.222.222 -t
```

6. List all your containers again. You can see all of the containers, including your new one with your customized name.

```
PS: node-0 Administrator> docker container ls --all

CONTAINER ID  IMAGE        COMMAND                  NAMES
e706e1168689  nanoserver   "ping 208.67.222.222…"   opendnsping
4bc814e2257b  nanoserver   "ping 8.8.4.4 -t"        eloquent_rubin
7aea600a8c76  nanoserver   "ping 8.8.8.8 -n 3"      frosty_mclean
```


7. Next, remove the exited container. To do this, use `docker container rm <container-id>`. In the example above, the Docker container ID is 7aea600a8c76.

```
PS: node-0 Administrator> docker container rm <container ID>

7aea600a8c76
```

8. Now try to remove one of the other Docker containers using the same command. It does not work. Why?

```
PS: node-0 Administrator> docker container rm <container ID>

Error response from daemon: You cannot remove a running container
4bc814e2257b2d1046c10e563b553279b7e11e8bf6309eff9047d2dfb086900f.
Stop the container before attempting removal or force remove
```

9. You can see that running containers are not removed. You'll have to look for an option to remove a running container. In order to find out the option you need to do a force remove, check the command line help. To do this with the `docker container rm` command, use the `--help` option:

```
PS: node-0 Administrator> docker container rm --help

Usage: docker container rm [OPTIONS] CONTAINER [CONTAINER...]

Remove one or more containers

Options:
 -f, --force     Force the removal of a running container (uses SIGKILL)
 -l, --link      Remove the specified link
 -v, --volumes   Remove the volumes associated with the container
```

> **Help works with all Docker commands** Not only can you use `--help` with `docker container rm`, but it works on all levels of `docker` commands. For example, `docker --help` provides you will all the available `docker` commands, as does `docker container --help` provide you with all available container commands.

10. Now, run a force remove on the running container you tried to remove in the two previous steps. This time it works.

```
PS: node-0 Administrator> docker container rm --force <container ID>

4bc814e2257b
```

11. Start another detached container pinging 8.8.8.8, with the name `pinggoogledns`.

```
PS: node-0 Administrator> docker container run --detach --name pinggoogledns `
    mcr.microsoft.com/windows/nanoserver:10.0.17763.737 ping 8.8.8.8 -t

38e121e629611daa0726a21d634bc5189400377d82882cc6fd8a3870dc9943a0
```

12. Now that you've finished your testing, you need to remove your containers. In order to remove all of them at once, you want to get only the container IDs. Look at `docker container ls --help` to get the information you need:

```
PS: node-0 Administrator> docker container ls --help

Usage:  docker container ls [OPTIONS]

List containers

Aliases:
  ls, ps, list

Options:
```



```
-a, --all           Show all containers (default shows just running)
-f, --filter filter Filter output based on conditions provided
--format string     Pretty-print containers using a Go template
-n, --last int      Show n last created containers (includes all states)
-l, --latest        Show the latest created container (includes all states)
    --no-trunc      Don't truncate output
-q, --quiet         Only display numeric IDs
-s, --size          Display total file sizes
```

13. To get only the container IDs, use the `--quiet` option. If you want to use only the container IDs of all existing containers to perform an action on, you can use `--quiet` with the `--all` option.

    ```
    PS: node-0 Administrator> docker container ls --all --quiet

    e706e1168689
    38e121e62961
    ```

14. Since we are done running pings on the public DNS servers, destroy the containers. To do this, use the syntax `docker container rm --force <containerID>`. However, this only kills one container at a time. We want to kill all the containers, no matter what state the containers are in. To get this information, you will need to use the output from `docker container ls --quiet --all`. To capture this output within the command, use `$(...)` to nest the listing command inside the `docker container rm` command.

    ```
    PS: node-0 Administrator> docker container rm --force `
        $(docker container ls --quiet --all)

    e706e1168689
    38e121e62961
    ```

1.1 Conclusion

This exercise taught you how to start, list, and kill containers. In this exercise you ran your first containers using `docker container run`, and how they are running commands inside the containers. You also learned to how to list your containers, and how to kill the containers using the command `docker container rm`. In you run into trouble, you've learned that the `--help` option can provide you with some ideas that could help get you answers.

2 Interactive Containers

By the end of this exercise, you should be able to:

- Launch an interactive shell in a new or existing container
- Run a child process inside a running container
- List containers using more options and filters

2.1 Writing to Containers

1. Create a container using the `mcr.microsoft.com/powershell:preview-nanoserver-1809` image, and connect to its powershell shell in interactive mode using the `-i` flag (also the `-t` flag, to request a TTY connection):

   ```
   PS: node-0 Administrator> docker container run `
       -it  mcr.microsoft.com/powershell:preview-nanoserver-1809
   ```

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

   ```
   PS C:\> ls
   PS C:\> cd .\Users\Public
   PS C:\Users\Public> echo 'hello world' > test.txt
   ```



```
PS C:\Users\Public> ls
PS C:\Users\Public> type .\test.txt
```

3. Exit the connection to the container:

```
PS C:Users\Public\> exit
```

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

```
PS: node-0 Administrator> docker container run `
    -it mcr.microsoft.com/powershell:preview-nanoserver-1809
```

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:

```
PS: node-0 Administrator> docker container ls --all

CONTAINER ID  IMAGE                     COMMAND       CREATED
041379c4f254  preview-nanoserver-1809  "pwsh.exe"  About a minute ago
d5b9286194f9  preview-nanoserver-1809  "pwsh.exe"  2 minutes ago
```

2. We can restart a container via the container ID listed in the first column. Use the container ID for the first `nanoserver` container you created with `powershell` as its command (see the `CREATED` column above to make sure you're choosing the *first* powershell container you ran):

```
PS: node-0 Administrator> docker container start <container ID>
PS: node-0 Administrator> docker container ls

CONTAINER ID  IMAGE                     COMMAND       CREATED        STATUS
d5b9286194f9  preview-nanoserver-1809  "pwsh.exe"  3 minutes ago  Up 2 seconds
```

   Your container status has changed from `Exited` to `Up`, via `docker container start`.

3. Now that your container is running again, launch a powershell process as a child process inside the container:

```
PS: node-0 Administrator> docker container exec -it <container ID> pwsh.exe
```

4. List the contents of the container's filesystem again with `ls`; your `test.txt` should be where you left it. Double check that its content is what you expect:

```
PS C:\> cd .\Users\Public
PS C:\Users\Public> ls
```

5. Exit the container again by typing `exit`.

2.3 Using Container Listing 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:

```
PS: node-0 Administrator> 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:

```
PS: node-0 Administrator> docker container ls -a -q
```

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


```
PS: node-0 Administrator> docker container ls -l
```

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

   ```
   PS: node-0 Administrator> docker container ls -a --filter "exited=0"
   ```

   The output of this command will list the containers that have exited successfully.

5. Clean up with:

   ```
   PS: node-0 Administrator> docker container rm -f $(docker container ls -aq)
   ```

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. 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

By the end of this exercise, you should be able to:

- Run a container detached from the terminal
- Fetch the logs of a container
- Attach a terminal to the STDOUT of a running container

3.1 Running a Container in the Background

1. First try running a container as usual; the STDOUT and STDERR streams from the main containerized process are directed to the terminal:

   ```
   PS: node-0 Administrator> docker container run `
       mcr.microsoft.com/windows/nanoserver:10.0.17763.737 ping 8.8.8.8 -n 2

   Pinging 8.8.8.8 with 32 bytes of data:
   Reply from 8.8.8.8: bytes=32 time=1ms TTL=113
   Reply from 8.8.8.8: bytes=32 time=1ms TTL=113

   Ping statistics for 8.8.8.8:
       Packets: Sent = 2, Received = 2, Lost = 0 (0% loss),
   Approximate round trip times in milli-seconds:
       Minimum = 1ms, Maximum = 1ms, Average = 1ms
   ```

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

   ```
   PS: node-0 Administrator> docker container run -d `
       mcr.microsoft.com/windows/nanoserver:10.0.17763.737 ping 8.8.8.8 -t

   5505012d74b3480a0a05ebd0ca1d256d08edc1916ef19d56fbe728bc3cecc502
   ```

   This time, we only see the container's ID; its STDOUT isn't being sent to the terminal.

3. Use this second container's ID to inspect the logs it generated:

   ```
   PS: node-0 Administrator> docker container logs <container ID>
   ```

   These logs correspond to STDOUT and STDERR from all processes running in the container. Also note when using container IDs: you don't need to specify the entire ID. Just enough characters from the start of the ID to uniquely identify it, often just 2 or 3, is sufficient.

3.2 Attaching to Container Output

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

   ```
   PS: node-0 Administrator> docker container attach <container ID>
   ```

2. We can leave attached mode by then pressing `CTRL+C`. Note that the container still happily runs in the background as you can confirm with `docker container ls`.

3.3 Using Logging Options

1. We saw previously how to read the entire log of a container's main process; 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:

   ```
   PS: node-0 Administrator> docker container logs --tail 5 <container ID>
   ```

   You should see the last 5 pings from this container.

3.4 Conclusion

In this exercise, we saw our first detached containers. Almost all containers you ever run will be running in detached mode; you can use `container attach` to interact with their main processes, as well as `container logs` to fetch their logs. Note that both `attach` and `logs` interact with the main process only - if you launch child processes inside a container, it's up to you to manage their STDOUT and STDERR streams.

4 Starting, Stopping, Inspecting and Deleting Containers

By the end of this exercise, you should be able to:

- Restart containers which have exited
- Distinguish between stopping and killing a container
- Fetch container metadata using `docker container inspect`
- Delete containers

4.1 Starting and Restarting Containers

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

   ```
   PS: node-0 Administrator> docker container run -d `
       --name demo mcr.microsoft.com/windows/nanoserver:10.0.17763.737 ping 8.8.8.8 -t
   PS: node-0 Administrator> docker container ls
   ```

   Note how we called the container `demo` for easier identification later on.

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

   ```
   PS: node-0 Administrator> docker container stop demo
   PS: node-0 Administrator> docker container ls -a
   ```

4.2 Inspecting a Container

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

   ```
   PS: node-0 Administrator> docker container start demo
   PS: node-0 Administrator> docker container inspect demo
   ```


You get a JSON object describing the container's config, metadata and state.

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 `select-string`:

   ```
   PS: node-0 Administrator> docker container inspect demo | select-string IPAddress
   ```

   The output should look similar to this:

   ```
   "SecondaryIPAddresses": null,
   "IPAddress": "",
           "IPAddress": "172.20.137.22",
   ```

3. Now try to use `select-string` for `Cmd`, the main process being run by this container. `select-string`'s simple text search doesn't always return helpful results:

   ```
   PS: node-0 Administrator> docker container inspect demo | select-string Cmd

   "Cmd": [
   ```

4. A more powerful 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 `select-string` for above, instead try:

   ```
   PS: node-0 Administrator> docker container inspect --format='{{.Config.Cmd}}' demo

   [ping 8.8.8.8 -t]
   ```

   This time, we get a the value of the `Config.Cmd` key from the `inspect` JSON.

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 selected previously.

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

   ```
   PS: node-0 Administrator> docker container inspect --format='{{json .Config}}' demo
   ```

   Try adding `| jq` to this command to get the same output a little bit easier to read.

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:

   ```
   PS: node-0 Administrator> docker container rm <container ID>
   PS: node-0 Administrator> 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:

   ```
   PS: node-0 Administrator> 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 exercise, you saw the basics of managing the container lifecycle. Containers can be restarted when stopped, and are only truly gone once they've been removed.

Also keep in mind the `docker container inspect` command we saw, for examining container metadata, state and config; this is often the first place to look when trying to troubleshoot a failed container.

5 Interactive Image Creation

By the end of this exercise, you should be able to:

- Capture a container's filesystem state as a new docker image
- Read and understand the output of `docker container diff`

5.1 Modifying a Container

1. Start a Powershell terminal in a Windows Server Core container:

   ```
   PS: node-0 Administrator> docker container run `
       -it --name demo mcr.microsoft.com/powershell:preview-windowsservercore-1809
   ```

2. Install a couple pieces of software in this container - First install a package manager; in this case **Chocolatey**:

   ```
   PS C:\> iex (iwr https://chocolatey.org/install.ps1 -UseBasicParsing)
   ```

   Then install some packages. There's nothing special about `wget`, any changes to the filesystem will do. Afterwards, exit the container:

   ```
   PS C:\> choco install -y wget
   PS C:\> exit
   ```

3. Finally, try `docker container diff` to see what's changed about a container relative to its image:

   ```
   PS: node-0 Administrator> docker container diff demo

   C Files
   C Files/Documents and Settings
   C Files/Program Files (x86)
   ...
   ```

   Those `C`s at the beginning of each line stand for files `C`hanged; lines that start with `D` indicate `D`eletions.

5.2 Capturing Container State as an Image

1. Installing wget 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`:

   ```
   PS: node-0 Administrator> docker container commit demo myapp:1.0
   ```

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

   ```
   PS: node-0 Administrator> docker image ls

   REPOSITORY    TAG      IMAGE ID            CREATED             SIZE
   myapp         1.0      9ce128f61c85        2 minutes ago       11GB
   ...
   ```

3. Create a container running Powershell using your new image, and check that wget is installed:

   ```
   PS: node-0 Administrator> docker container run -it myapp:1.0 powershell
   PS C:\> cd \ProgramData\chocolatey\lib
   PS C:\> ls

       Directory: C:\ProgramData\chocolatey\lib
   ```



```
Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        8/29/2018  10:30 PM                chocolatey
d-----        8/29/2018  10:31 PM                Wget
```

The software you installed in your previous container is also available in this container, and all subsequent containers you start from the image you captured using `docker container commit`.

5.3 Conclusion

In this exercise, you saw how to inspect the contents of a container's read / write layer with `docker container diff`, and commit those changes to a new image layer with `docker container commit`. Committing a container as an image in this fashion can be useful when developing an environment inside a container, when you want to capture that environment for reproduction elsewhere.

6 Creating Images with Dockerfiles (1/2)

By the end of this exercise, you should be able to:

- Write a Dockerfile using the `FROM` and `RUN` commands
- Build an image from a Dockerfile
- Anticipate which image layers will be fetched from the cache at build time
- Fetch build history for an image

6.1 Writing and Building a Dockerfile

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

```
FROM mcr.microsoft.com/windows/servercore:10.0.17763.805
SHELL ["powershell", "-Command"]
RUN iex (invoke-webrequest https://chocolatey.org/install.ps1 -UseBasicParsing)
RUN choco install -y wget
```

2. Build your image with the `build` command. Don't miss the `.` at the end; that's the path to your `Dockerfile`. Since we're currently in the directory `myimage` which contains it, the path is just `.` (here).

   ```
   PS: node-0 myimage> docker image build -t myimage .
   ```

   You'll see a long build output - we'll go through the meaning of this output in a demo later. For now, everything is good if it ends with `Successfully tagged myimage:latest`.

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, e.g.:

   ```
   PS: node-0 myimage> docker container run -it myimage powershell
   PS C:\> wget https://chocolatey.org -UseBasicParsing -o index.html
   PS C:\> cat index.html
   PS C:\> exit
   ```

   You should see the HTML from chocolatey.org, downloaded by `wget` from within your container.

4. It's also possible to pipe a Dockerfile in from STDIN; try rebuilding your image with the following:

   ```
   PS: node-0 myimage> cat Dockerfile | docker build -t myimage -f - .
   ```

   (This is useful when reading a Dockerfile from a remote location with `Invoke-WebRequest`, 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`.

```
FROM mcr.microsoft.com/windows/servercore:10.0.14393.2972
SHELL ["powershell", "-Command"]
RUN iex (invoke-webrequest https://chocolatey.org/install.ps1 -UseBasicParsing)
RUN choco install -y wget
RUN choco install -y 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:

```
FROM mcr.microsoft.com/windows/servercore:10.0.14393.2972
SHELL ["powershell", "-Command"]
RUN iex (invoke-webrequest https://chocolatey.org/install.ps1 -UseBasicParsing)
RUN choco install -y vim
RUN choco install -y wget
```

   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:

   `PS: node-0 myimage> docker image history myimage`

   Note the image id of the layer built for chocolatey install.

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

```
...
RUN choco 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)

By the end of this exercise, you should be able to:

- Define a default process for an image to containerize by using the `ENTRYPOINT` or `CMD` Dockerfile commands
- Understand the differences and interactions between `ENTRYPOINT` and `CMD`

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", "-n", "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:

```
PS: node-0 myimage> docker image build -t myimage .
```

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

```
PS: node-0 myimage> docker container run myimage
```

   You should see the command provided by the `CMD` parameter in the Dockerfile running.

4. Try explicitly providing a command when running a container:

```
PS: node-0 myimage> docker container run myimage powershell write-host "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:

```
PS: node-0 myimage> docker image build -t myimage .
PS: node-0 myimage> docker container run myimage
```

   You'll get an error. What went wrong?

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

```
PS: node-0 myimage> docker container run myimage 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", "-n", "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 the defaults:

```
PS: node-0 myimage> docker image build -t myimage .
PS: node-0 myimage> docker container run myimage
```

   You should see it pinging the default IP, `127.0.0.1`.

4. Run another container with a custom IP argument:

```
PS: node-0 myimage> docker container run myimage 8.8.8.8
```

   This time, you should see a ping to `8.8.8.8`. 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 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

By the end of this exercise, you should be able to:

- Write a Dockerfile that describes multiple images, which can copy files from one image to the next.

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.go` to the `multi-stage` folder containing **Hello World** in Go:

```
package main

import "fmt"

func main() {
    fmt.Println("hello world")
}
```

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

```
FROM golang:1.12.5-windowsservercore
COPY . /code
WORKDIR /code
RUN go build hello.go
CMD ["\\code\\hello.exe"]
```

4. Build the image and observe its size:

```
PS: node-0 multi-stage> docker image build -t my-app-large .
PS: node-0 multi-stage> docker image ls | select-string my-app-large

REPOSITORY     TAG      IMAGE ID      CREATED         SIZE
my-app-large   latest   7c95f4e0112e  11 minutes ago  5.86GB
```

5. Test the image to confirm it actually works:

```
PS: node-0 multi-stage> docker container run my-app-large
```

It should print "hello world" in the console.

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

```
FROM golang:1.12.5-windowsservercore AS gobuild
COPY . /code
WORKDIR /code
RUN go build hello.go

FROM mcr.microsoft.com/windows/nanoserver:10.0.17763.737
COPY --from=gobuild /code/hello.exe /hello.exe
CMD ["\\hello.exe"]
```


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

```
PS: node-0 multi-stage> docker image build -t my-app-small .
PS: node-0 multi-stage> docker image ls | select-string 'my-app-'

REPOSITORY     TAG      IMAGE ID      CREATED         SIZE
my-app-small   latest   13a42c43f45f  11 minutes ago  253MB
my-app-large   latest   7c95f4e0112e  13 minutes ago  5.86GB
```

   As expected, the size of the multi-stage build is much smaller than the large one since it does not contain the .NET SDK.

8. Finally, make sure the app actually works:

```
PS: node-0 multi-stage> docker container run my-app-small
```

   You should get the expected 'hello world' output from the container with just the required executable.

8.2 Building Intermediate Images

In the previous step, we took our compiled executable from the first build stage, but that image wasn't tagged as a regular image we can use to start containers with; only the final `FROM` statement generated a tagged 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:

```
PS: node-0 multi-stage> docker image build -t my-build-stage --target gobuild .
```

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

```
PS: node-0 multi-stage> docker container run -it --rm my-build-stage hello.exe
```

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, just like how you currently probably have separate build and run environments for your software. This will result in containers that start faster, and are less vulnerable to attack.

9 Managing Images

By the end of this exercise, you should be able to:

- Rename and retag an image
- Push and pull images from the public registry
- Delete image tags and image layers, and understand the difference between the two operations

9.1 Making an Account on Docker's Hosted Registry

1. If you don't have one already, head over to https://hub.docker.com and make an account. For the rest of this workshop, `<Docker ID>` refers to the username you chose for this account.

9.2 Tagging and Listing Images

1. Download the `mcr.microsoft.com/windows/nanoserver:10.0.17763.737` image from Docker Hub:

   ```
   PS: node-0 Administrator> docker image pull `
       mcr.microsoft.com/windows/nanoserver:10.0.17763.737
   ```

2. Make a new tag of this image:

   ```
   PS: node-0 Administrator> docker image tag `
       mcr.microsoft.com/windows/nanoserver:10.0.17763.737 mynanoserver:dev
   ```

   Note no new image has been created; `mynanoserver:dev` is just a pointer pointing to the same image as `mcr.microsoft.com/windows/nanoserver:10.0.17763.737`.

3. List your images:

   ```
   PS: node-0 Administrator> docker image ls

   ...
   mynanoserver       dev             4c872414bf9d   3 weeks ago   250MB
   windows/nanoserver 10.0.17763.737 4c872414bf9d   3 weeks ago   250MB
   ...
   ```

   You should have `mcr.microsoft.com/windows/nanoserver:10.0.17763.737` and `mynanoserver:dev` both listed, but they ought to have the same hash under image ID, since they're actually the same image. (Note you'll have a lot of other images, too - these were pre-downloaded for this workshop. On your own machines, you'll have to download the images you want using `docker image pull` like above).

9.3 Sharing Images on Docker Hub

1. Push your image to Docker Hub:

   ```
   PS: node-0 Administrator> docker image push mynanoserver:dev
   ```

   You should get an `denied: requested access to the resource is denied` error.

2. 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 `<Docker ID>/<repo name>[:<optional tag>]`.

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

   ```
   PS: node-0 Administrator> $user="<Docker ID>"
   PS: node-0 Administrator> docker image tag mynanoserver:dev $user/mynanoserver:dev
   PS: node-0 Administrator> docker image push $user/mynanoserver:dev
   ```

4. Search Docker Hub for your new `<Docker ID>/mynanoserver` repo, and confirm that you can see the `:dev` tag therein.

5. Next, write a Dockerfile that uses `$user/mynanoserver:dev` as its base image, and add an `ENTRYPOINT` or `CMD` parameter. Build the image, and simultaneously tag it as `:1.0`:

   ```
   PS: node-0 Administrator> docker image build -t $user/mynanoserver:1.0 .
   ```

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

7. 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`:

   ```
   PS: node-0 Administrator> docker image rm mynanoserver:dev
   ```

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

9.4 Conclusion

In this exercise, we practiced tagging images and exchanging them on the public registry. The namespacing rules for images on registries are *mandatory*: user-generated images to be exchanged on the public registry must be named like `<Docker ID>/<repo name>[:<optional tag>]`; official images on Hub just have the repo name and tag.

Also note that as we saw when building images, image names and tags are just pointers; deleting an image with `docker image rm` just deletes that pointer if the corresponding image layers are still being referenced by another such pointer. Only when the last pointer is deleted are the image layers actually destroyed by `docker image rm`.

10 Database Volumes

By the end of this exercise, you should be able to:

- Provide a docker volume as a database backing to mongodb
- Make one mongodb container's database available to other mongodb containers

10.1 Launching mongodb

1. Download a mongodb image, and inspect it to determine its default volume usage:

   ```
   PS: node-0 Administrator> docker image pull training/mongo:ws19
   PS: node-0 Administrator> docker image inspect training/mongo:ws19

   ...
   "Volumes": {
       "C:\\data\\configdb": {},
       "C:\\data\\db": {}
   },
   ...
   ```

   You should see a `Volumes` block like the above, indicating that those paths in the container filesystem will get volumes automatically mounted to them when a container is started based on this image.

2. Set up a running instance of this mongodb container:

   ```
   PS: node-0 Administrator> docker container run --name mongoserv -d `
       -v mongodata:C:\data\db `
       training/mongo:ws19
   ```

   Notice the explicit volume mount, `-v mongodata:C:\data\db`; if we hadn't done this, a randomly named volume would have been mounted to the container's `C:\data\db`. Naming the volume explicitly is a best practice that will become useful when we start mounting this volume in multiple containers.

10.2 Writing to the Database

1. Spawn a `mongo` process inside your mongodb container:

   ```
   PS: node-0 Administrator> docker container exec -it mongoserv mongo
   ```

   You'll be presented with a mongodb terminal where you can manipulate a database directly.

2. Create an arbitrary table in the database:

   ```
   > use products
   > db.products.save({"name":"widget", "price":"18.95"})
   > db.products.save({"name":"sprocket", "price":"1.45"})
   ```

   Double check you created the table you expected, and then quit this container:


```
> db.products.find()

{ "_id" : ObjectId("..."), "name" : "widget", "price" : "18.95" }
{ "_id" : ObjectId("..."), "name" : "sprocket", "price" : "1.45" }

> exit
```

3. Delete the `mongoserv` container:

   ```
   PS: node-0 Administrator> docker container rm -f mongoserv
   ```

4. Create a new mongodb server container, mounting the `mongodata` volume just like last time:

   ```
   PS: node-0 Administrator> docker container run --name mongoserv -d `
       -v mongodata:C:\data\db `
       training/mongo:ws19
   ```

5. Spawn another `mongo` process inside this new container to get a mongodb terminal, also like before:

   ```
   PS: node-0 Administrator> docker container exec -it mongoserv mongo
   ```

6. List the contents of the `products` database:

   ```
   > use products
   > db.products.find()
   ```

   The contents of the database have survived the deletion and recreation of the database container; this would not have been true if the database was keeping its data in the writable container layer. As above, use `exit` to quit from the mongodb prompt.

7. Delete your mongodb container and volume:

   ```
   PS: node-0 Administrator> docker container rm -f mongoserv
   PS: node-0 Administrator> docker volume rm mongodata
   ```

10.3 Conclusion

Whenever data needs to live longer than the lifecycle of a container, it should be pushed out to a volume outside the container's filesystem; numerous popular databases are containerized using this pattern.

11 Introduction to Container Networking

By the end of this exercise, you should be able to:

- Attach containers to Docker's default nat network
- Resolve containers by DNS entry

11.1 Inspecting the Default Nat Network

1. Let's use the Docker CLI to inspect the NAT network. The `docker network inspect` command yields network information about what containers are connected to the specified network; the default network is always called `nat`, so run:

   ```
   PS: node-1 Administrator> docker network inspect nat
   ```

   This returns state and metadata about your network; note especially the list of containers attached to this network:

   ```
   "Containers": {}
   ```

   Currently, there are no containers attached to the `nat` network; but if you create any without specifying a network, this is where they'll be attached by default.

11.2 Connecting Containers to Default Nat

1. Start some named containers:

```
PS: node-1 Administrator> docker container run --name=u1 -dt `
    mcr.microsoft.com/powershell:preview-nanoserver-1809
PS: node-1 Administrator> docker container run --name=u2 -dt `
    mcr.microsoft.com/powershell:preview-nanoserver-1809
```

2. Inspect the `nat` network again:

```
PS: node-1 Administrator> docker network inspect nat
```

You should see two new entries in the `Containers` section of the result, one for each container:

```
...
"Containers": {
    "45e8576...": {
        "Name": "u1",
        "EndpointID": "8e938af....",
        "MacAddress": "00:15:5d:e6:0a:ec",
        "IPv4Address": "172.20.131.137/16",
        "IPv6Address": ""
    },
    "b7e49f5...": {
        "Name": "u2",
        "EndpointID": "266f0c0...",
        "MacAddress": "00:15:5d:e6:07:06",
        "IPv4Address": "172.20.135.21/16",
        "IPv6Address": ""
    },
    ...
}
...
```

We can see that each container gets a `MacAddress` and an `IPv4Address` associated. The `nat` network is providing level 2 routing and transfers network packets between MAC addresses.

3. Connect to container u2 of your containers using `docker container exec -it u2 pwsh.exe`.
4. 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 both the IP and the container name.
5. Clean up these containers:

```
PS: node-1 Administrator> docker container rm -f u1 u2
```

11.3 Conclusion

In this exercise, you explored the most basic example of container networking: two containers communicating on the same host via network address translation and a layer 2 in-software router in the form a a Hyper-V switch. In addition to this basic routing technology, you saw how Docker leverages DNS lookup via container name to make our container networking portable; by allowing us to reach another container purely by name, without doing any other service discovery, we make it simple to design application logic meant to communicate container-to-container. At no point did our application logic need to discover anything directly about the networking infrastructure it was running on.

12 Container Port Mapping

By the end of this exercise, you should be able to:

- Forward traffic from a port on the docker host to a port inside a container's network namespace
- Define ports to automatically expose in a Dockerfile

12.1 Port Mapping at Runtime

1. Run an IIS container with no special port mappings:

   ```
   PS: node-1 Administrator> docker container run -d microsoft/iis
   ```

   IIS stands up a landing page at `http://<IP>:80`; try to visit this at your host's public IP, and it won't be visible; no external traffic can make it past the Windows NAT's firewall to the container running IIS.

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

   ```
   PS: node-1 Administrator> docker container run -d -p 5000:80 --name iis microsoft/iis
   ```

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

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

   ```
   PS: node-1 Administrator> docker container port iis
   ```

4. Open a browser window and visit your IIS landing page at `<host ip>:5000`; you should see the default IIS landing page. Your browser's network request to port 5000 in the host network namespace has been forwarded on to the container's network namespace at port 80.

12.2 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 `portdemo`, create a Dockerfile:

   ```
   FROM microsoft/iis
   EXPOSE 80
   ```

2. Build your image as `my_iis`:

   ```
   PS: node-1 portdemo> docker image build -t my_iis .
   ```

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

   ```
   PS: node-1 portdemo> docker container run -d -P my_iis
   ```

4. Use `docker container ls` or `docker container port` to find out what host ports were used, and visit your IIS landing page in a browser at `<node-1 public IP>:<port>`.

5. Clean up your containers:

   ```
   PS: node-1 portdemo> docker container rm -f $(docker container ls -aq)
   ```

12.3 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.

13 Creating a Swarm

By the end of this exercise, you should be able to:

- Create a swarm in high availability mode
- Set default address pools
- Check necessary connectivity between swarm nodes

- Configure the swarm's TLS certificate rotation

13.1 Starting Swarm Mode

1. On node-0, initialize swarm and create a cluster with a default address pool for a discontiguous address range of 10.85.0.0/16 and 10.91.0.0/16 with a default subnet size of 128 addresses. This will be your first manager node:

```
PS: node-0 Administrator> $PRIVATEIP = '<node-0 private IP>'
PS: node-0 Administrator> docker swarm init `
    --advertise-addr ${PRIVATEIP} `
    --listen-addr ${PRIVATEIP}:2377 `
    --default-addr-pool 10.85.0.0/16 `
    --default-addr-pool 10.91.0.0/16 `
    --default-addr-pool-mask-length 25

Swarm initialized: current node (xyz) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-0s96... 10.10.1.40:2377

To add a manager to this swarm, run 'docker swarm join-token manager'
    and follow the instructions.
```

2. Confirm that Swarm Mode is active and that the default address pool configuration has been registered by inspecting the output of:

```
PS: node-0 Administrator> docker system info

...
Swarm: active
...
    Default Address Pool: 10.85.0.0/16  10.91.0.0/16
    SubnetSize: 25
...
```

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

```
PS: node-0 Administrator> docker node ls

ID      HOSTNAME   STATUS   AVAILABILITY   MANAGER STATUS
xyz *   node-0     Ready    Active         Leader
```

   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:

```
PS: node-0 Administrator> 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.

13.2 Adding 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:

```
PS: node-0 Administrator> docker swarm join-token worker
```


2. Switch into `node-1` from the dropdown menu, and paste the result of the last step there. This new node will join the swarm as a worker.
3. Do `docker node ls` on `node-0` 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.
4. Have a look at open TCP connections on `node-0`:

```
PS: node-0 Administrator> netstat

Active Connections

  Proto  Local Address          Foreign Address        State
  TCP    10.10.5.199:2377       ip-10-10-21-25:52054   ESTABLISHED
  TCP    10.10.5.199:3389       WimaxUser37230-226:63128  CLOSE_WAIT
  TCP    10.10.5.199:3389       ool-944be43f:51428     ESTABLISHED
  TCP    10.10.5.199:7946       ip-10-10-21-25:52010   TIME_WAIT
  TCP    10.10.5.199:7946       ip-10-10-21-25:52057   TIME_WAIT
  TCP    10.10.5.199:51233      13.89.217.116:https    ESTABLISHED
  TCP    10.10.5.199:51862      52.94.233.129:https    TIME_WAIT
  TCP    10.10.5.199:51865      ip-10-10-21-25:7946    TIME_WAIT
  TCP    10.10.5.199:51866      ip-10-10-21-25:7946    TIME_WAIT
```

   You should see something similar; most notably, the first line in the example above corresponds to `node-1` (private IP 10.10.21.25 in this example) connecting to this node on tcp/2377. Also, many mutual connections are made between the two nodes on 7946, corresponding to gossip control plane communications.
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:

```
PS: node-0 Administrator> docker node ls

ID      HOSTNAME   STATUS   AVAILABILITY   MANAGER STATUS
ghi     node-3     Ready    Active
def     node-2     Ready    Active
abc     node-1     Ready    Active
xyz *   node-0     Ready    Active         Leader
```

13.3 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 using a raft consensus of multiple managers to preserve swarm state.

1. Promote two of your workers to manager status by executing, on `node-0`:

```
PS: node-0 Administrator> docker node promote node-1 node-2
```

2. Finally, do a `docker node ls` to check and see that you now have three managers; `node-1` and `node-2` should report as `Reachable`, indicating that they are healthy members of the raft consensus. Note that manager nodes also count as worker nodes - tasks can still be scheduled on them as normal by default.

13.4 Conclusion

In this exercise, you set up a basic high-availability swarm. In practice, it is crucial to have at least 3 (and always an odd number) of managers in order to ensure high availability of your cluster, and to ensure that the management, control, and data plane communications a swarm maintains can proceed unimpeded between all nodes.

14 Starting a Service

By the end of this exercise, you should be able to:

- Schedule a docker service across a swarm
- Predict and understand the scoping behavior of docker overlay networks
- Scale a service on swarm up or down

14.1 Creating an Overlay Network and Service

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

   ```
   PS: node-0 Administrator> docker network create --driver overlay my_overlay
   ```

2. Verify that the network subnet was taken from the address pool defined when creating your swarm::

   ```
   PS: node-0 Administrator> docker network inspect my_overlay

   ...
   "Subnet": "10.85.0.0/25",
   "Gateway": "10.85.0.1"
   ...
   ```

   The overlay network has been assigned a subnet from the address pool we specified when creating our swarm.

3. Create a service featuring a `training/probe`, which sends a periodic network request to docker.com to see if the internet is reachable from your container:

   ```
   PS: node-0 Administrator> docker service create --name prober `
       --network my_overlay training/probe:ws19
   ```

   Note the syntax is a lot like `docker container run`; an image (`training/probe:ws19`) is specified after some flags, which conainerizes a default process.

4. Get some information about the currently running services:

   ```
   PS: node-0 Administrator> docker service ls

   ID             NAME     MODE         REPLICAS   IMAGE
   k6caojtyqp2p   prober   replicated   1/1        training/probe:ws19
   ```

5. Check which node the container was created on:

   ```
   PS: node-0 Administrator> docker service ps prober

   ID       NAME       IMAGE                       NODE     DESIRED STATE   ...
   jot...   prober.1   docker service ps prober   node-1   Running         ...
   ```

   In my case, the one container we started for this service was scheduled on `node-1`.

6. Scale up the number of concurrent tasks that our `prober` service is running to 3:

   ```
   PS: node-0 Administrator> docker service update prober --replicas=3

   prober
   overall progress: 3 out of 3 tasks
   1/3: running   [==================================================>]
   2/3: running   [==================================================>]
   3/3: running   [==================================================>]
   verify: Service converged
   ```

7. Now run `docker service ps prober` to inspect the service. How were tasks distributed across your swarm?

8. Run `docker network inspect my_overlay` on any node that has a `prober` task scheduled on it. Look for the `Containers` key in the output; it indicates the containers on this node attached to the `my_overlay` network. Also, look for the `Peers` list; mine looks like:

```
"Peers": [
    {
        "Name": "967da1a0349f",
        "IP": "10.10.12.136"
    },
    {
        "Name": "d1cd9f4a25bb",
        "IP": "10.10.35.36"
    },
    {
        "Name": "d7e00d4376ca",
        "IP": "10.10.57.19"
    }
]
```

Challenge: Looking at your own `Peers` list, what do the IPs correspond to?

14.2 Inspecting Service Logs

1. Manager nodes can assemble all logs for all tasks of a given service:

   ```
   PS: node-0 Administrator> docker service logs --tail 100 prober
   ```

   The last 100 lines of logs for all 3 probe containers will be displayed.

2. If instead you'd like to see the logs of a single task, on a manager node run `docker service ps prober`, choose any task ID, and run `docker service logs <task ID>`.

14.3 Cleanup

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

   ```
   PS: node-0 Administrator> docker service rm $(docker service ls -q)
   ```

14.4 Conclusion

In this exercise, we saw the basics of creating, scheduling and updating services. A common mistake people make is thinking that a service is just the containers scheduled by the service; in fact, a Docker service is the definition of *desired state* for those containers. Changing a service definition does not in general change containers directly; it causes them to get rescheduled by Swarm in order to match their new desired state.

15 Node Failure Recovery

By the end of this exercise, you should be able to:

- Anticipate swarm scheduling decisions when nodes fail and recover
- Force swarm to reallocate workload across a swarm

15.1 Setting up a Service

1. Set up an `microsoft/iis` service with four replicas on `node-0`, and wait for all four tasks to be up and running:


```
PS: node-0 Administrator> docker service create --replicas 4 --name iis `
    microsoft/iis
```

15.2 Simulating Node Failure

1. Switch to the non-manager node in your swarm (node-3), and simulate a node failure by rebooting it:

   ```
   PS: node-3 Administrator> Restart-Computer -Force
   ```

2. Back on node-0, keep doing `docker service ps iis` every few seconds; what happens to the task running on the rebooted node? Look at its desired state, any other tasks that get scheduled with the same name, and keep watching until node-3 comes back online.

15.3 Force Rebalancing

By default, if a node fails and rejoins a swarm it *will not* get its old workload back; if we want to redistribute workload across a swarm after new nodes join (or old nodes rejoin), we need to force-rebalance our tasks.

1. Make sure node-3 has fully rebooted and rejoined the swarm.

2. Force rebalance the tasks:

   ```
   PS: node-0 Administrator> docker service update --force iis
   ```

3. After the service converges, check which nodes the service tasks are scheduled on:

   ```
   PS: node-0 Administrator>docker service ps iis

   ID      NAME       IMAGE                NODE    DESIRED STATE  CURRENT STATE
   dv5...  iis.1      microsoft/iis  node-0  Running        Running 20 seconds ago
   xge...   \_ iis.1  microsoft/iis  node-0  Shutdown       Shutdown 36 seconds ago
   jma...  iis.2      microsoft/iis  node-1  Running        Running about a minute ago
   afd...   \_ iis.2  microsoft/iis  node-1  Shutdown       Shutdown about a minute ago
   3hc...  iis.3      microsoft/iis  node-2  Running        Running 39 seconds ago
   j30...   \_ iis.3  microsoft/iis  node-2  Shutdown       Shutdown 56 seconds ago
   yzz...  iis.4      microsoft/iis  node-3  Running        Running 58 seconds ago
   w9s...   \_ iis.4  microsoft/iis  node-2  Shutdown       Shutdown about a minute ago
   bqz...   \_ iis.4  microsoft/iis  node-3  Shutdown       Shutdown 3 minutes ago
   ```

   The _ shape indicate *ancestor* tasks which have been shut down and replaced by a new task, typically after reconfiguring the service or rebalancing like we've done here. Once the rebalance is complete, the current tasks for the iis service should be evenly distributed across your swarm.

15.4 Cleanup

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

   ```
   PS: node-0 Administrator> docker service rm $(docker service ls -q)
   ```

15.5 Conclusion

In this exercise, you saw swarm's scheduler in action - when a node is lost from the swarm, tasks are automatically rescheduled to restore the state of our services. Note that nodes joining or rejoining the swarm do not get workload automatically reallocated from existing nodes to them; rescheduling only happens when tasks crash, services are first scheduled, or you force a reschedule as above.

16 Swarm Scheduling

By default, the Swarm scheduling algorithm tries to spread workload out roughly evenly across your Swarm. In many cases, we want to exert more nuanced control over what containers get scheduled where, in order to respect resource availability, hardware requirements, application high availability and other concerns. By the end of this exercise, you should be able to:

- Impose memory resource reservations and limitations
- Schedule services in global or replicated mode, and define appropriate use cases for each
- Schedule tasks on a subset of nodes via label constraints
- Schedule topology-aware services

16.1 Restricting Resource Consumption

By default, containers can consume as much compute resources as they want. Overconsumption of memory can cause an entire node to fail; as such, we need to make sure we're scheduling only as many containers on a host as that host can realistically support, and preventing them from allocating too much memory.

1. On your worker node `node-3` (**not** on a manager node please), open the Task Manager, either through the search bar or by typing `taskmgr` in the command prompt, and click on **More Details** to get a report of current compute resource consumption.
2. Still on that same node, spin up a container that will allocate as much memory as it can:

   ```
   PS: node-3 Administrator> docker container run `
       training/winstress:ws19 pwsh.exe C:\saturate-mem.ps1
   ```

   After a minute, the memory column in Task Manager will max out, and your node will become unresponsive. Use `CTRL+C` to detach from the container, and `docker container rm -f` to remove this container.
3. If we spun this container up as a service, it could take down our entire swarm. Prevent this from happening by imposing a memory limitation on the service; from `node-0`:

   ```
   PS: node-0 Administrator> docker service create `
       --replicas 4 --limit-memory 4096M --name memcap `
       training/winstress:ws19 pwsh.exe C:\saturate-mem.ps1
   ```

   Observe the memory consumption through Task Manager on any node hosting a task for this service; it'll spike, but be capped before it can completely take down the node.
4. Observe your container's memory consumption directly with `docker stats` on any node hosting one of these containers:

   ```
   docker stats

   CONTAINER ID  NAME             CPU %   PRIV WORKING SET ...
   09fd6cef8528  memcap.1.rko...  48.29%  2.065GiB         ...
   ```

   The private working set memory is capped below whatever limit you set with the `--limit-memory` flag. Use `CTRL+C` to break out of the `docker stats` view when done.
5. Clean up by deleting your service:

   ```
   PS: node-0 Administrator> docker service rm memcap
   ```
6. So far, we've limited the amount of resources a container can consume once scheduled, but we still haven't prevented the scheduler from *overprovisioning* containers on a node; we would like to prevent Docker from scheduling more containers on a node than that node can support. Delete and recreate your service one more time, this time also imposing a scheduling reservation with `--reserve-memory`:

   ```
   PS: node-0 Administrator> docker service create `
       --replicas 4 --limit-memory 2048M --name memcap `
       --reserve-memory 2048M `
       training/winstress:ws19 pwsh.exe C:\saturate-mem.ps1
   ```



```
PS: node-0 Administrator> docker service ps memcap
```

You should see your four tasks running happily.

7. Now scale up your service to more replicas than your current cluster can support:

```
PS: node-0 Administrator> docker service update memcap --replicas=10 --detach
PS: node-0 Administrator> docker service ps memcap

ID        NAME        CURRENT STATE            ERROR
...
z6t...    memcap.5    Pending 2 seconds ago    "no suitable node (insufficien…"
xf3...    memcap.6    Pending 2 seconds ago    "no suitable node (insufficien…"
...
```

Many of your tasks should be stuck in `CURRENT STATE: Pending`. Inspect one of them:

```
PS: node-0 Administrator> docker inspect <task ID>


...
    "Status": {
        "Timestamp": "2019-01-29T16:31:45.5859483Z",
        "State": "pending",
        "Message": "pending task scheduling",
        "Err": "no suitable node (insufficient resources on 4 nodes)",
        "PortStatus": {}
    }
...
```

From the `Status` block in the task info, we can see that this task isn't getting scheduled because there isn't sufficient resources available in the cluster to support it.

> **Always limit resource consumption** using *both* limits and reservations. Failing to do so is a very common mistake that can lead to a widespread cluster outage; if your cluster is running at near-capacity and one node fails, its workload will get rescheduled to other nodes, potentially causing them to also fail, initiating a cascading cluster outage.

8. Clean up by removing your service:

```
PS: node-0 Administrator> docker service rm memcap
```

16.2 Configuring Global Scheduling

So far, all the services we've created have been run in the default *replicated mode*; as we've seen, Swarm spreads containers out across the cluster, potentially respecting resource reservations. Sometimes, we want to run services that create *exactly one* container on each host in our cluster; this is typically used for deploying daemon, like logging, monitoring, or node management tools which need to run locally on every node.

1. Create a globally scheduled service:

```
PS: node-0 Administrator> docker service create --mode global `
    --name my-global `
    mcr.microsoft.com/windows/nanoserver:10.0.17763.737 ping 8.8.8.8 -t
```

2. Check what nodes your service tasks were scheduled on:

```
PS: node-0 Administrator> docker service ps my-global

ID             NAME                      IMAGE                        NODE
rjl91n1i5o4g   competent_taussig.j9h...  nanoserver:10.0.17763.737    node-3
pzkiv2kpsu26   competent_taussig.c3v...  nanoserver:10.0.17763.737    node-2
k767q7i1f73t   competent_taussig.afo...  nanoserver:10.0.17763.737    node-1
```



```
wem26fmzq2k5  competent_taussig.lnk...  nanoserver:10.0.17763.737  node-0
```

One task is scheduled on every node in the swarm; as you add or remove nodes from the swarm, the global service will be rescaled appropriately.

3. Remove your service with `docker service rm my-global`.

16.3 Scheduling via Node Constraints

Sometimes, we want to confine our containers to specific nodes; for this, we can use *constraints* and *node properties*.

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

```
PS: node-0 Administrator> docker node update --label-add datacenter=east node-0
PS: node-0 Administrator> docker node update --label-add datacenter=east node-1
```

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

```
PS: node-0 Administrator> docker node update --label-add datacenter=west node-2
PS: node-0 Administrator> docker node update --label-add datacenter=west node-3
```

Note these labels are user-defined; `datacenter` and its values `east` and `west` can be anything you like.

3. Schedule a service constrained to run on nodes labeled as `datacenter=east`:

```
PS: node-0 Administrator> docker service create --replicas 4 `
    --constraint node.labels.datacenter==east `
    --name east-deploy `
    mcr.microsoft.com/windows/nanoserver:10.0.17763.737 ping 8.8.8.8 -t
```

4. Check what nodes your tasks were scheduled on as above; they should all be on nodes bearing the `datacenter==east` label (`node-0` and `node-1`).

5. Remove your service, and schedule another, this time constrained to run only on worker nodes:

```
PS: node-0 Administrator> docker service rm east-deploy
PS: node-0 Administrator> docker service create --replicas 4 `
    --constraint node.role==worker `
    --name worker-only `
    mcr.microsoft.com/windows/nanoserver:10.0.17763.737 ping 8.8.8.8 -t
```

Once again, check where your `worker-only` service tasks got scheduled; they'll all be on `node-3`, your only worker.

> **Keep workload off of managers** using these selectors, especially in production. If something goes badly wrong with a workload container and causes its host to crash, we don't want to take down a manager and possibly lose our raft consensus. As we've seen, Swarm can recover from losing workers automatically; a lost manager consensus can be much harder to recover from.

6. Clean up by removing this service: `docker service rm worker-only`.

16.4 Scheduling Topology-Aware Services

Oftentimes, we want to schedule workload to be tolerant of faults in our datacenters; we wouldn't want every replica for a service on one power zone or one rack which can go down all at once, for example.

1. Create a service using the `--placement-pref` flag to spread replicas across our `datacenter` label:

```
PS: node-0 Administrator> docker service create --name spreaddemo `
    --replicas=2 --publish 8000:80 `
    --placement-pref spread=node.labels.datacenter `
    mcr.microsoft.com/windows/nanoserver:10.0.17763.737 ping 8.8.8.8 -t
```

There should be `nanoserver` containers present on nodes with every possible value of the `node.labels.datacenter` label, one in `datacenter=east` nodes, and one in `datacenter=west` nodes.

2. Use `docker service ps spreaddemo` as above to check that replicas got spread across the datacenter labels.
3. Clean up: `docker service rm spreaddemo`.

16.5 Conclusion

In this exercise, we saw how to use resource allocations, global scheduling, labels and node properties to influence scheduling. A few best practices:

- **Always apply memory and CPU limits and reservations**, and in essentially all cases, the limit should be less than or equal to the reservation to ensure nodes are never overprovisioned.
- **Keep workload off of manager** as mentioned above, to prevent application logic bugs from taking down your manager consensus
- **Don't overconstrain your scheduler**: it can be tempting to exert strict control over exactly what gets scheduled exactly where; don't. If a service is constrained to run on a very small set of nodes, and those nodes go down, the service will become unschedulable and suffer an outage. Let your orchestraor's scheduler make decisions as independently as possible in order to maximize workload resilience.

17 Provisioning Swarm Configuration

When deploying an application, especially one meant to be migrated across different environments, it's helpful to be able to provision configuration like environment variables and config files to your services in a modular, pluggable fashion. By the end of this exercise, you should be able to:

- Assemble application components together as a Docker stack
- Provision insecure configuration to service containers using `.env` files and Docker configs
- Provision secure configuration to service containers using Docker secrets

17.1 Creating a Stack

So far, we've run individual services with `docker service create`. As we build more complex applications consisting of multiple components, we'd like a way to capture them all in a single file we can version control and recreate; for this, we can use *stack files*.

1. Create a file called `mystack.yaml` with the following content:

```
version: "3.7"

services:
  whoami:
    image: training/whoami-windows:ws19
    deploy:
      mode: global
    ports:
      - target: 5000
        published: 8080
```

   This stack file will create a single service named `whoami-windows`, based on the `training/whoami-windows:ws19` image, schedule it globally, and make the `whoami` response reachable on port 8080 of any host in the cluster.

   > Docker stack file syntax is based on Docker Compose; we'll see numerous examples of this syntax in this workshop, but if you'd like the full reference, see the docs at https://dockr.ly/2iHUpeX.

2. Deploy your stack:

   ```
   PS: node-0 Administrator> docker stack deploy -c mystack.yaml stackdemo
   ```

   Your service is created, along with a default network for the stack (more on service networking in a future exercise).

3. List your stacks and, see its services:

```
PS: node-0 Administrator> docker stack ls

NAME                SERVICES            ORCHESTRATOR
stackdemo           1                   Swarm

PS: node-0 Administrator> docker service ls

ID      NAME              MODE    REPLICAS  IMAGE
bon...  stackdemo_whoami  global  4/4       training/whoami-windows:ws19
```

   By default, your service gets named as the stack (`stackdemo`), concatenated with the key you labeled your service with in your stack file (`whoami`).

4. Make sure everything is working as expected by visiting the Who Am I response at `http://<public IP>:8080`, where `<public IP>` is the public IP of any node in your swarm - by default, Swarm uses its *layer 4 mesh net* to route request arriving at the exposed port (8080) on *any* host in the swarm to the appropriate backend containers.

17.2 Defining and Using Docker Configs

Above, we created a simple website with four replicas; we'd like to set up a load balancer to direct traffic to our website replicas, but we don't want to have to create a special load balancer image just for this one task; we'd rather use a generic load balancer image, and provision it with the appropriate config at startup. For this we can use a *docker config*.

1. Create an nginx configuration file called `nginx.conf`, changing the lines with `<node-x public IP>` to the public IPs of each of your swarm nodes:

```
worker_processes  1;

events {
    worker_connections  1024;
}

stream {
    upstream myapp {
        server <node-0 public IP>:8080;
        server <node-1 public IP>:8080;
        server <node-2 public IP>:8080;
        server <node-3 public IP>:8080;
    }

    server {
        listen 80;
        proxy_pass myapp;
    }
}
```

2. Modify your stack file to add in a proxy service, using this config file as a Docker config:

```
version: "3.7"

services:
  whoami:
    image: training/whoami-windows:ws19
    deploy:
      mode: global
```



```
    ports:
      - target: 5000
        published: 8080

  proxy:
    image: training/win-nginx:ee3.0-ws19
    ports:
      - target: 80
        published: 8001
    configs:
      - source: nginxconf
        target: C:\nginx\nginx-1.12.0\conf\nginx.conf

configs:
  nginxconf:
    file: .\nginx.conf
```

Here we've added a new top level object, `configs`, that lists Docker config objects (just `nginxconf` in this example), which each specify a file to be populated by. In our `proxy` service we mount this config by name, and give it a path to mount to.

3. Update your stack:

```
PS: node-0 Administrator> docker stack deploy -c mystack.yaml stackdemo
```

Note this is the exact same command you used to create the stack in the first place; recreating an existing stack will apply only the updates since your last deploy.

4. List and inspect your `config` objects:

```
PS: node-0 Administrator> docker config ls

ID                      NAME                   CREATED              UPDATED
9k8qm1en5e7t5tn7q...    stackdemo_nginxconf    About a minute ago   About a minute ago

PS: node-0 Administrator> docker config inspect --pretty <config ID>

ID:                     9k8qm1en5e7t5tn7qyloi2pf0
Name:                   stackdemo_nginxconf
Labels:
 - com.docker.stack.namespace=stackdemo
Created at:             2019-02-03 00:44:37.9298585 +0000 utc
Updated at:             2019-02-03 00:44:37.9298585 +0000 utc
Data:
worker_processes  1;

events {
    worker_connections  1024;
}

stream {
    upstream myapp {
        server 3.92.18.65:8080;
        server 3.87.220.140:8080;
        server 3.80.99.91:8080;
        server 3.90.223.203:8080;
    }

    server {
        listen 80;
```



```
        proxy_pass myapp;
    }
}
```

We can recover the plain-text contents of any config option in this manner.

5. Make sure this all worked as expected by visiting port 8001 (the public port for your `nginx` service) and make sure you can see the `whoami` response, proving your proxy is routing to your simple website being served on 8080.
6. Clean up: `docker stack rm stackdemo`

17.3 Defining and Using .env Files and Secrets

Above, we provisioned an entire configuration file to a service via a docker config object. Often we only want to provision individual tokens, like paths or passwords; furthermore, these tokens can have varying security needs. For non-secure information, we can specify *environment variables* in our containers, and for sensitive information we should use *docker secrets*, as follows.

1. Create a new directory `image-secrets` on `node-0` 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]))
```

For optimal security, secret information like passwords shouldn't be stored in an environment variable directly; Docker will provision the secret to the container as a file. We can then define an environment variable that points at the path of this secret file, which our script can then consume.

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

```
FROM python:3.8.0-windowsservercore-1809
RUN mkdir -p /app
WORKDIR /app
COPY . /app
CMD python ./app.py; sleep 100000
```

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

```
PS: node-0 image-secrets> docker image build -t <Docker ID>/secrets-demo:1.0 .
PS: node-0 image-secrets> docker image push <Docker ID>/secrets-demo:1.0
```

4. Next let's create a secret. In the current directory create a file called `password.txt` and add the value `my-pass` to it. Turn the contents of that file into a docker secret:

```
PS: node-0 image-secrets> docker secret create mypass ./password.txt
PS: node-0 image-secrets> rm password.txt
```

Remember to delete the plaintext copy of your password in `password.txt`. Swarm encrypts your secret value and won't return it in plain text, so with this file removed your secret is secure at rest on your management cluster.

5. Define your non-secure environment variables in a file called `myconf.env`:

```
PASSWORD_FILE=C:\ProgramData\Docker\secrets\mypass
```


By default, Docker will place a secret called `mypass` at the path `C:\ProgramData\Docker\secrets\mypass`, which we're going to inform our containerized process of via and environment variable defined in this env file.

6. Create a stack file called `secretstack.yaml` that makes a service out of your `secrets-demo:1.0` image, and provisions it with your secret password and environment variable (don't forget to change `<Docker ID>` to your Docker Hub ID):

```
version: "3.7"

services:
  myapp:
    image: <Docker ID>/secrets-demo:1.0
    env_file:
      - myconf.env
    secrets:
      - mypass

secrets:
  mypass:
    external: true
```

Here we introduce a few keys:

- `services:env_file` lists files that contain key/value pairs like our `myconf.env` to be declared as environment variables in the containers for this service
- `services:secrets` lists docker secrets, created as above, to provision to the containers for this service. By default, the content of the secret will be available as a file at the path `C:\ProgramData\Docker\secrets\<secret name>`
- The top level `secrets` key lists secrets created as above, for consumption in your services.

7. Deploy your stack, list the services on you swarm, and get the logs for your single service:

```
PS: node-0 image-secrets> docker stack deploy -c .\secretstack.yaml secretdemo
Creating network secretdemo_default
Creating service secretdemo_destination

PS: node-0 image-secrets>docker service ls

ID            NAME              MODE        REPLICAS  IMAGE
8aoeanv90dqm  secretdemo_myapp  replicated  1/1       training/secrets-demo:1.0

PS: node-0 image-secrets> docker service logs <service ID>

secretdemo_myapp.1.xxx@node-0  | ***** Docker Secrets ******
secretdemo_myapp.1.xxx@node-0  | USERNAME: ContainerAdministrator
secretdemo_myapp.1.xxx@node-0  | PASSWORD_FILE: C:\ProgramData\Docker\secrets\mypass
secretdemo_myapp.1.xxx@node-0  | PASSWORD: my-pass
```

If all has been successful, your script will have used the environment variable `PASSWORD_FILE` to locate your secret password, and read it from there. Of course this is just a toy script to demonstrate usage, but the same pattern of provisioning secure information through secrets pointed at by environment variables is a common best practice for handling this type of config.

8. Optional: Locate the node running your single container for this service, and use `docker container inspect` on it. Notice all the environment variables defined in the container are visible in the `Env:` block of the output. A common mistake when provisioning configuration is to provide passwords directly as environment variables; do that, and those passwords will be exposed in plain text to anyone who has inspect access to your containers.

9. Clean up: `docker stack rm secretdemo`

17.4 Conclusion

In this exercise, we saw several different methods for defining and provisioning configurations, as well as a few examples of stack files for defining and composing all the elements of our application. Deciding what information to provision via configurations is an important architectural choice; in general, anything that's going to change when moving from environment to environment is a good candidate for a config, since env files, docker configs, and docker secrets are all modular and defined separately from the service definition itself; by separating configs in this way, we can just swap the config out when changing environments, without redefining our services. The (usually worse) alternative to provisioning by config is to include this information directly in the image; this is a good choice for information that is the same in all environments you plan on running that image in, but can lead to image management complexity and loss of portability if environment-specific information is hard-coded into the image. Of course, secure information like passwords should *never* be hard-coded into images; they should strictly be provisioned as Docker secrets, and consumed only from the filesystem inside the container to which they are mounted.

18 Routing Traffic to Docker Services

By the end of this exercise, you should be able to:

- Route traffic to a Docker service from within the swarm using dnsrr DNS lookups
- Route traffic to a Docker service from outside the swarm using host port publishing

18.1 Routing Cluster-Internal Traffic

1. By *cluster-internal traffic*, we mean traffic from originating from a container running on your swarm, sending a request to another container running on the same swarm. Let's create a stack with two such services, attached to a custom overlay network; create a file `net-demo.yaml` with the following content:

```
version: "3.7"

services:
  destination:
    image: training/whoami-windows:ws19
    deploy:
      replicas: 3
    networks:
      - demonet

  origin:
    image: mcr.microsoft.com/powershell:preview-nanoserver-1809
    command: ["pwsh.exe", "-Command", "Start-Sleep", "100000"]
    networks:
      - demonet

networks:
  demonet:
    driver: overlay
```

   Here our `destination` service is using the `deploy:replicas` key to ask for three replica containers based on the `training/whoami-windows` image; this image serves a simple web page on port 5000 that reports the ID of the container its running in. Our `origin` service is using the `command` key to define what process to run in a container based on the `mcr.microsoft.com/powershell:preview-nanoserver-1809` image; in this case we just put it to sleep so we have a running container we can attach to and interact with later.

2. Deploy your stack, and find out what node your `origin` container is running on:

```
PS: node-0 Administrator> docker stack deploy -c net-demo.yaml netstack
```



```
PS: node-0 Administrator> docker stack ps netstack

ID       NAME                     IMAGE                         NODE     DESIRED STATE
73i...   netstack_destination.1   training/whoami-windows       node-2   Running
y8i...   netstack_origin.1        nanoserver:10.0.14393.2551    node-0   Running
pgw...   netstack_destination.2   training/whoami-windows       node-1   Running
thi...   netstack_destination.3   training/whoami-windows       node-3   Running
```

3. Connect to the node running your `origin` container (`node-0` in the example above), and create a powershell inside that container:

```
PS: node-0 Administrator> docker container ls

CONTAINER ID        IMAGE                         COMMAND
3047dbb47a7a        nanoserver:10.0.14393.2551    Start-Sleep...

PS: node-0 Administrator> docker container exec -it <container ID> pwsh.exe

PS C:\>
```

4. Probe the DNS resolution of your `destination` service:

```
PS C:\> [System.Net.Dns]::GetHostEntry('destination')

HostName    Aliases AddressList
--------    ------- -----------
destination {}      {10.85.6.130}
```

   By default, swarm services are assigned a *virtual IP*, and their service name will DNS resolve to this VIP inside any container attached to the same Docker network. Try doing `docker service inspect netstack_destination` back on a manager to confirm that the IP you resolved above is what's listed for the virtual IP for this service.

5. Try contacting your `destination` service using the VIP you found above:

```
PS C:\> (Invoke-WebRequest http://<virtual IP>:5000).Content

I am 02FF20A20861
```

   Execute the same a few more times to see swarm's default load balancing: subsequent requests get routed across all `whoami` containers in round robin fashion.

6. Clean up by removing your stack: `docker stack rm netstack`.

18.2 Routing Cluster-External Traffic

In the last section, we routed traffic from one Docker service to another within the same swarm. If we want to route ingress traffic from an external network to a service, we have to expose it on the *routing mesh*.

1. Create a new stack file `mesh.yaml`:

```yaml
version: "3.7"

services:
  destination:
    image: training/whoami-windows:ws19
    deploy:
      replicas: 3
    ports:
      - 8080:5000
```


Here the `ports:` key is specifying that traffic arriving at port 8080 on *any node in the swarm* should be forwarded to port 5000 of our `whoami` container.

2. Deploy your stack:

```
PS: node-0 Administrator> docker stack deploy -c mesh.yaml mesh
```

3. List your service and observe which nodes the containers got scheduled on:

```
PS: node-0 Administrator> docker service ls
PS: node-0 Administrator> docker service ps mesh_destination

ID                  NAME                  IMAGE                  NODE
1wsqyjgznv0a        mesh_destination.1    whoami-windows:ws19    node-0
ljhr1dukx6v0        mesh_destination.2    whoami-windows:ws19    node-3
albbml1idsw4        mesh_destination.3    whoami-windows:ws19    node-2
```

4. Visit the public IP of every one of your nodes on port 8080 in your browser; no matter which public IP you choose, your request will get routed to a `whoami` backend container - even if you visit the public IP of a node *not* running a `whoami` container. This is the mesh net in action (note that many browsers' caching behavior will hide the round robin load balancing; if you want to see the load balancing in the browser, try using a 'history-less' browser mode, closing and reopening the browser or completely purging history between each refresh).
5. Clean up by deleting your `mesh` stack.

18.3 Conclusion

In this exercise, you saw the basic service discovery and routing that Swarm enables via DNS lookup of service names, VIP routing and port forwarding across network namespaces. In general, using this networking plane to do our service discovery helps make our inter-service communication more robust against container failure. If we were communicating with a container directly by its IP, we would have to constantly monitor whether that container was still actually reachable at that IP; Swarm effectively does that for us when we communicate via DNS resolution of service names to VIPs, since a failed container will get pulled out of the virtual IP server's round-robin routing automatically. In the case of using the mesh net to route ingress traffic from outside our cluster, mapping the external port on *every* host in the swarm onto the service's VIP means our external load balancers (which is realistically the point of ingress for external traffic to our swarm), doesn't need to know anything about where our service is scheduled; it can simply send requests to any node in the swarm, and Docker handles the rest.

19 Updating Applications

Once we have defined an application as a Docker stack, we will periodically want to update its scale, configuration, and source code. By the end of this exercise, you should be able to:

- Scale up microservice components of a stack to improve application performance
- Define and trigger a rolling update of a service
- Define and trigger an automatic rollback of a failed service update

19.1 Deploying Dockercoins

For this exercise, we'll work with a toy application called *Dockercoins*. This application is a toy 'dockercoin' miner, consisting of five microservices interacting in the following workflow:

1. A **worker** container requests a random number from a random number generator **rng**
2. After receiving a random number, the worker pushes it to a **hasher** container, which computes a hash of this number.
3. If the hash of the random number starts with 0, we accept this as a Dockercoin, and forward it to a **redis** container.

4. Meanwhile, a **webui** container monitors the rate of coins being sent to redis, and visualizes this as a chart on a web page.
5. Download the Dockercoins app from Github and change directory to ~/orchestration-workshop-net:

   ```
   PS: node-0 Administrator> git clone -b ee3.0-ws19 `
       https://github.com/docker-training/orchestration-workshop-net.git

   PS: node-0 Administrator> cd ~/orchestration-workshop-net
   ```

6. The Dockercoins application is defined in `stack.yml`; have a look at this file, and make sure you understand what every key is doing. Once you're satisfied with this, deploy the stack:

   ```
   PS: node-0 orchestration-workshop-net> docker stack deploy `
       -c stack.yml dockercoins
   ```

   Visit the Dockercoins web frontend at <public IP>:8000, where <public IP> is the public IP of any node in your swarm. You should see Dockercoins getting mined at a rate of a few per second.

19.2 Scaling Up an Application

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 `stack.yml` to set the number of replicas to create using the `deploy` and `replicas` keys:

   ```
   worker:
     image: training/dc_worker:ws19
     networks:
       - dockercoins
     deploy:
       replicas: 2
   ```

2. Update your app by running the same command you used to launch it in the first place:

   ```
   PS: node-0 orchestration-workshop-net> docker stack deploy -c `
       stack.yml dockercoins
   ```

   Check the web frontend; after a few seconds, you should see about double the number of hashes per second, as expected.

3. 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; let's investigate the two services `worker` is interacting with: `rng` and `hasher`.
4. First, we need to expose ports for the `rng` and `hasher` services, so we can probe their latency. Update their definitions in `stack.yml` with a `ports` key:

   ```
   rng:
     image: training/dc_rng:ws19
     networks:
       - dockercoins
     ports:
       - target: 80
         published: 8001

   hasher:
     image: training/dc_hasher:ws19
     networks:
       - dockercoins
   ```



```
  ports:
    - target: 80
      published: 8002
```

Update the services by redeploying the stack file:

```
PS: node-0 orchestration-workshop-net> docker stack deploy `
    -c stack.yml dockercoins
```

If this is successful, a `docker service ls` should show `rng` and `hasher` exposed on the appropriate ports.

5. With `rng` and `hasher` exposed, we can use `httping` to probe their latency; in both cases, `<public IP>` is the public IP of any node in your swarm:

```
PS: node-0 orchestration-workshop-net> httping -c 5 <public IP>:8001
PS: node-0 orchestration-workshop-net> httping -c 5 <public IP>: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).

6. Modify the definition of our `rng` service in `stack.yml` to be globally scheduled:

```
rng:
  image: training/dc_rng:ws19
  networks:
    - dockercoins
  deploy:
    mode: global
  ports:
    - target: 80
      published: 8001
```

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

```
PS: node-0 orchestration-workshop-net> docker stack rm dockercoins
PS: node-0 orchestration-workshop-net> docker stack deploy `
    -c stack.yml dockercoins
```

8. Check the web frontend again; you should finally see the factor of 10 improvement in performance versus a single worker container, from 3-4 coins per second to around 35.

19.3 Creating Rolling Updates

Beyond scaling up an existing application, we'll periodically want to update the underlying source code of one or more of our components; Swarm provides mechanisms for rolling out updates in a controlled fashion that minimizes downtime.

1. First, let's change one of our services a bit: open `orchestration-workshop-net/worker/Program.cs` in your favorite text editor, and find the following section:

```
private static void WorkOnce(){
    Console.WriteLine("Doing one unit of work");
    Thread.Sleep(100);  // 100 ms
```

Change the 100 to a 10. Save the file, exit the text editor.

2. Rebuild the worker image with a tag of `<Docker ID>/dc_worker:ws19-1.1`, and push it to Docker Hub.

3. Change the `image:` value for the `worker` service in your `stack.yml` file to that of the image you just pushed.

4. Start the update:

```
PS: node-0 orchestration-workshop-net> docker stack deploy `
    -c stack.yml dockercoins
```

Use `docker stack ps dockercoins` every couple of seconds to watch tasks get updated to our new 1.1 image one at a time.

19.4 Parallelizing Updates

1. We can also set our updates to run in batches by configuring some options associated with each service. Change the update parallelism to 2 and the delay to 5 seconds on the `worker` service by editing its definition in `stack.yml`:

```
worker:
  image: training/dc_worker:ws19
  networks:
    - dockercoins
  deploy:
    replicas: 10
    update_config:
      parallelism: 2
      delay: 5s
```

2. Roll back the `worker` service to its original image:

```
PS: node-0 orchestration-workshop-net> docker stack deploy `
    -c stack.yml dockercoins
```

Run `docker service ps dockercoins_worker` every couple of seconds; you should see pairs of worker tasks getting shut down and replaced with the original version, with a 5 second delay between updates (this is perhaps easiest to notice by examining the `NAME` column - every worker replica will start with one dead task from when you upgraded in the last step; you should be able to notice pairs of tasks with two dead ancestors as this rollback moves through the list, two at a time).

19.5 Auto-Rollback Failed Updates

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:

```
worker:
  image: training/dc_worker:ws19
  networks:
    - dockercoins
  deploy:
    replicas: 10
    update_config:
      parallelism: 2
      delay: 5s
      failure_action: rollback
      max_failure_ratio: 0.2
      monitor: 20s
```

These parameters will trigger a rollback if more than 20% of services tasks fail in the first 20 seconds after an update.

2. Update your stack to make sure the rollback parameters are in place before you attempt to update your image:


```
PS: node-0 orchestration-workshop-net> docker stack deploy `
    -c stack.yml dockercoins
```

3. Make a broken version of the `worker` service to trigger a rollback with; try removing all the `using` commands at the top of `worker/Program.cs`, for example. Then rebuild the worker image with a tag `<Docker ID>/dc_worker:bugged`, push it to Docker Hub, and attempt to update your service:

```
PS: node-0 orchestration-workshop-net> docker image build `
    -t <Docker ID>/dc_worker:bugged worker
PS: node-0 orchestration-workshop-net> docker image push `
    <Docker ID>/dc_worker:bugged
```

4. Update your `stack.yml` file to use the `:bugged` tag for the `worker` service, and redeploy your stack as above.

5. Use `docker stack ps dockercoins` to watch the `:bugged` tag getting deployed, failing, and rolling back automatically over the next minute or two:

```
NAME                       IMAGE              CURRENT STATE
dockercoins_worker.1       dc_worker:ws19     Running 2 minutes ago
dockercoins_worker.2       dc_worker:ws19     Running 2 minutes ago
dockercoins_worker.3       dc_worker:ws19     Running about a minute ago
 \_ dockercoins_worker.3   dc_worker:bugged   Failed about a minute ago
 \_ dockercoins_worker.3   dc_worker:bugged   Failed about a minute ago
 \_ dockercoins_worker.3   dc_worker:bugged   Failed about a minute ago
dockercoins_worker.4       dc_worker:ws19     Running about a minute ago
 \_ dockercoins_worker.4   dc_worker:bugged   Failed about a minute ago
dockercoins_worker.5       dc_worker:ws19     Running about a minute ago
 \_ dockercoins_worker.5   dc_worker:bugged   Failed about a minute ago
 \_ dockercoins_worker.5   dc_worker:bugged   Failed about a minute ago
 \_ dockercoins_worker.5   dc_worker:bugged   Failed about a minute ago
dockercoins_worker.6       dc_worker:ws19     Running 2 minutes ago
dockercoins_worker.7       dc_worker:ws19     Running 2 minutes ago
dockercoins_worker.8       dc_worker:ws19     Running 2 minutes ago
dockercoins_worker.9       dc_worker:ws19     Running about a minute ago
 \_ dockercoins_worker.9   dc_worker:bugged   Failed about a minute ago
dockercoins_worker.10      dc_worker:ws19     Running 2 minutes ago
```

   For example, this table indicates that tasks 3, 4, 5 and 9 all attempted to update to the `:bugged` tag, failed, and successfully rolled back to the `:ws19` tag (the `\_` symbol is meant to indicate failed ancestors for an individual task; so `dockercoins_worker.3` above made three failed attempts to run the `:bugged` image before rolling back).

6. Clean up by removing your stack:

```
PS: node-0 orchestration-workshop-net> docker stack rm dockercoins
```

19.6 Optional Challenge: Improving Dockercoins

Dockercoins' stack file is very rudimentary, and not at all suitable for production. If you have time, try modifying Dockercoins' stack file with some of the best practices you've learned in this workshop; think about things like security, operational stability, latency and scheduling. This activity is best done in groups! Partner up with someone else and discuss what improvements you can make, then try them out and make sure they work as you expected.

19.7 Conclusion

In this exercise, we explored deploying and redeploying an application as stacks and services. Note that relaunching a running stack updates all the objects it manages in the most non-disruptive way possible; there is usually no need to remove a stack before updating it. In production, rollback contingencies should always be used to cautiously upgrade images, cutting off potential damage before an entire service is taken down.

20 Installing Kubernetes

By the end of this exercise, you should be able to:

- Set up a Kubernetes cluster with one master and one node

20.1 Initializing Kubernetes

1. On kube-0, initialize the cluster with kubeadm:

```
[centos@kube-0 ~]$ sudo kubeadm init --pod-network-cidr=192.168.0.0/16 \
    --ignore-preflight-errors=SystemVerification
```

   If successful, the output will end with a join command:

```
...
You can now join any number of machines by running the following on each node
as root:

  kubeadm join 10.10.29.54:6443 --token xxx --discovery-token-ca-cert-hash sha256:yyy
```

2. To start using you cluster, you need to run:

```
[centos@kube-0 ~]$ mkdir -p $HOME/.kube
[centos@kube-0 ~]$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
[centos@kube-0 ~]$ sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

3. List all your nodes in the cluster:

```
[centos@kube-0 ~]$ kubectl get nodes
```

   Which should output something like:

```
NAME      STATUS     ROLES     AGE       VERSION
kube-0    NotReady   master    2h        v1.11.1
```

   The NotReady status indicates that we must install a network for our cluster.

4. Let's install the Calico network driver:

```
[centos@kube-0 ~]$ kubectl apply -f https://bit.ly/2v9yaaV
```

5. After a moment, if we list nodes again, ours should be ready:

```
[centos@kube-0 ~]$ kubectl get nodes -w
NAME      STATUS     ROLES     AGE       VERSION
kube-0    NotReady   master    1m        v1.11.1
kube-0    NotReady   master    1m        v1.11.1
kube-0    NotReady   master    1m        v1.11.1
kube-0    Ready     master    2m        v1.11.1
kube-0    Ready     master    2m        v1.11.1
```

6. Execute the join command you found above when initializing Kubernetes on kube-1 (you'll need to add sudo to the start, and --ignore-preflight-errors=SystemVerification to the end), and then check the status back on kube-0:

```
[centos@kube-1 ~]$ sudo kubeadm join ... --ignore-preflight-errors=SystemVerification
[centos@kube-0 ~]$ kubectl get nodes
```

   After a few moments, there should be two nodes listed - all with the Ready status.

7. Let's see what system pods are running on our cluster:


```
[centos@kube-0 ~]$ kubectl get pods -n kube-system
```

which results in something similar to this:

```
NAME                                    READY     STATUS    RESTARTS   AGE
calico-etcd-pfhj4                       1/1       Running   1          5h
calico-kube-controllers-559c657d6d-ztk8c   1/1       Running   1          5h
calico-node-89k9v                       2/2       Running   0          4h
calico-node-brqxz                       2/2       Running   2          5
coredns-78fcdf6894-gtj87                1/1       Running   1          5h
coredns-78fcdf6894-nz2kw                1/1       Running   1          5h
etcd-kube-0                             1/1       Running   1          5h
kube-apiserver-kube-0                   1/1       Running   1          5h
kube-controller-manager-kube-0          1/1       Running   1          5h
kube-proxy-vgrtm                        1/1       Running   0          4h
kube-proxy-ws2z5                        1/1       Running   0          5h
kube-scheduler-kube-0                   1/1       Running   1          5h
```

We can see the pods running on the master: etcd, api-server, controller manager and scheduler, as well as calico and DNS infrastructure pods deployed when we installed calico.

20.2 Conclusion

At this point, we have a Kubernetes cluster with one master and one worker ready to accept workloads.

21 Kubernetes Orchestration

By the end of this exercise, you should be able to:

- Define and launch basic pods, replicaSets and deployments using `kubectl`
- Get metadata, configuration and state information about a kubernetes object using `kubectl describe`
- Update an image for a pod in a running kubernetes deployment

21.1 Creating Pods

1. On your master node, create a yaml file `pod.yaml` to describe a simple pod with the following content:

```
apiVersion: v1
kind: Pod
metadata:
  name: demo
spec:
  containers:
  - name: nginx
    image: nginx:1.7.9
```

2. Deploy your pod:

```
[centos@kube-0 ~]$ kubectl create -f pod.yaml
```

3. Confirm your pod is running:

```
[centos@kube-0 ~]$ kubectl get pod demo
```

4. Get some metadata about your pod:


```
[centos@kube-0 ~]$ kubectl describe pod demo
```

5. Delete your pod:

```
[centos@kube-0 ~]$ kubectl delete pod demo
```

6. Modify `pod.yaml` to create a second container inside your pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: demo
spec:
  containers:
  - name: nginx
    image: nginx:1.7.9
  - name: sidecar
    image: centos:7
    command: ["ping"]
    args: ["8.8.8.8"]
```

7. Deploy this new pod, and create a bash shell inside the container named `sidecar`:

```
[centos@kube-0 ~]$ kubectl create -f pod.yaml
[centos@kube-0 ~]$ kubectl exec -c=sidecar -it demo -- /bin/bash
```

8. From within the `sidecar` container, fetch the nginx landing page on the default port 80 using `localhost`:

```
[root@demo /]# curl localhost:80
```

You should see the html of the nginx landing page. Note **these containers can reach each other on localhost**, meaning they are sharing a network namespace. Now list the processes in your `sidecar` container:

```
[root@demo /]# ps -aux
```

You should see the `ping` process we containerized, the shell we created to explore this container using `kubectl exec`, and the `ps` process itself - but no `nginx`. While a network namespace is shared between the containers, they still have their own PID namespace (for example).

9. Finally, remember to exit out of this pod, and delete it:

```
[root@demo /]# exit
[centos@kube-0 ~]$ kubectl delete pod demo
```

21.2 Creating ReplicaSets

1. On your master node, create a yaml file `replicaset.yaml` to describe a simple replicaSet with the following content:

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: rs-demo
spec:
  replicas: 3
  selector:
    matchLabels:
      component: reverse-proxy
  template:
    metadata:
      labels:
```



```
        component: reverse-proxy
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
```

Notice especially the `replicas` key, which defines how many copies of this pod to create, and the `template` section; this defines the pod to replicate, and is described almost exactly like the first pod definition we created above. The difference here is the required presence of the `labels` key in the pod's metadata, which must match the `selector -> matchLabels` item in the specification of the replicaSet.

2. Deploy your replicaSet, and get some state information about it:

```
[centos@kube-0 ~]$ kubectl create -f replicaset.yaml
[centos@kube-0 ~]$ kubectl describe replicaset rs-demo
```

After a few moments, you should see something like

```
Name:         rs-demo
Namespace:    default
Selector:     component=reverse-proxy
Labels:       component=reverse-proxy
Annotations:  <none>
Replicas:     3 current / 3 desired
Pods Status:  3 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:  component=reverse-proxy
  Containers:
   nginx:
    Image:        nginx:1.7.9
    Port:         <none>
    Host Port:    <none>
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Events:
  Type    Reason            Age   From                   Message
  ----    ------            ----  ----                   -------
  Normal  SuccessfulCreate  35s   replicaset-controller  Created pod: rs-demo-jxmjj
  Normal  SuccessfulCreate  35s   replicaset-controller  Created pod: rs-demo-dmdtf
  Normal  SuccessfulCreate  35s   replicaset-controller  Created pod: rs-demo-j62fx
```

Note the replicaSet has created three pods as requested, and will reschedule them if they exit.

3. Try killing off one of your pods, and reexamining the output of the above `describe` command. The `<pod name>` comes from the last three lines in the output above, such as `rs-demo-jxmjj`:

```
[centos@kube-0 ~]$ kubectl delete pod <pod name>
[centos@kube-0 ~]$ kubectl describe replicaset rs-demo
```

The dead pod gets rescheduled by the replicaSet, similar to a failed task in Docker Swarm.

4. Delete your replicaSet:

```
[centos@kube-0 ~]$ kubectl delete replicaset rs-demo
```

21.3 Creating Deployments

1. On your master node, create a yaml file `deployment.yaml` to describe a simple deployment with the following content:


```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
```

Notice this is the exact same structure as your replicaSet yaml above, but this time the `kind` is `Deployment`. Deployments create a replicaSet of pods, but add some deployment management functionality on top of them, such as rolling updates and rollback.

2. Spin up your deployment, and get some state information:

```
[centos@kube-0 ~]$ kubectl create -f deployment.yaml
[centos@kube-0 ~]$ kubectl describe deployment nginx-deployment
```

The `describe` command should return something like:

```
Name:                   nginx-deployment
Namespace:              default
CreationTimestamp:      Thu, 24 May 2018 04:29:18 +0000
Labels:                 <none>
Annotations:            deployment.kubernetes.io/revision=1
Selector:               app=nginx
Replicas:               3 desired | 3 updated | 3 total | 3 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=nginx
  Containers:
   nginx:
    Image:        nginx:1.7.9
    Port:         <none>
    Host Port:    <none>
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Available      True    MinimumReplicasAvailable
  Progressing    True    NewReplicaSetAvailable
OldReplicaSets:  <none>
NewReplicaSet:   nginx-deployment-85f7784776 (3/3 replicas created)
Events:
  Type    Reason             Age   From                   Message
```



```
  ----    ------              ----  ----                    -------
  Normal  ScalingReplicaSet   10s   deployment-controller  Scaled up replica set
                                                           nginx-deployment-85f7784776
                                                           to 3
```

Note the very last line, indicating this deployment actually created a replicaSet which it used to scale up to three pods.

3. List your replicaSets and pods:

```
[centos@kube-0 ~]$ kubectl get replicaSet
[centos@kube-0 ~]$ kubectl get pod
```

You should see one replicaSet and three pods created by your deployment.

4. Upgrade the nginx image from `1.7.9` to `1.9.1`:

```
[centos@kube-0 ~]$ kubectl set image deployment/nginx-deployment nginx=nginx:1.9.1
```

5. After a few seconds, `kubectl describe` your deployment as above again. You should see that the image has been updated, and that the old replicaSet has been scaled down to 0 replicas, while a new replicaSet (with your updated image) has been scaled up to 3 pods. List your replicaSets one more time:

```
[centos@kube-0 ~]$ kubectl get replicaSets
```

You should see something like

```
NAME                          DESIRED   CURRENT   READY     AGE
nginx-deployment-69df9ccbf8   3         3         3         4m
nginx-deployment-85f7784776   0         0         0         9m
```

Do a `kubectl describe replicaSet <replicaSet scaled down to 0>`; you should see that while no pods are running for this replicaSet, the old replicaSet's definition is still around so we can easily roll back to this version of the app if we need to.

6. Clean up your cluster:

```
[centos@kube-0 ~]$ kubectl delete deployment nginx-deployment
```

21.4 Conclusion

In this exercise, you explored the basic scheduling objects of pods, replicaSets, and deployments. Each object is responsible for a different part of the orchestration stack; pods are the basic unit of scheduling, replicaSets do keep-alive and scaling, and deployments provide update and rollback functionality. In a sense, these objects all 'nest' one inside the next; by creating a deployment, you implicitly created a replicaSet which in turn created the corresponding pods. In most cases, you're better off creating deployments rather than replicaSets or pods directly; this way, you get all the orchestrating scheduling features you would expect in analogy to a Docker Swarm service.

22 Provisioning Kube Configuration

By the end of this exercise, you should be able to:

- write kube yaml describing secrets and config maps, and associate them with pods and deployments.

22.1 Provisioning ConfigMaps

1. Create a file on `kube-0` called `env-config` with the following content:

```
user=moby
db=mydb
```


2. Create a configMap that parses each line in this file as a separate key/value pair:

```
[centos@kube-0 ~]$ kubectl create configmap dbconfig --from-env-file=env-config
```

3. Ask kubectl to repeat this configMap back to us, in yaml:

```
[centos@kube-0 ~]$ kubectl get configmap dbconfig -o yaml

apiVersion: v1
data:
  db: mydb
  user: moby
kind: ConfigMap
metadata:
  creationTimestamp: "2019-01-08T15:41:31Z"
  name: dbconfig
  namespace: default
  resourceVersion: "709821"
  selfLink: /api/v1/namespaces/default/configmaps/dbconfig
  uid: df14e7a7-135b-11e9-87ee-0242ac11000a
```

The key/value pairs parsed from `env-config` are visible under the `data` key in this file; we could have created the same configMap declaratively via `kubectl create -f <yaml filename>` like we've been doing so far, using this yaml.

> **Imperative vs. Declarative kubectl commands**: `kubectl` supports two syntaxes for most actions: *imperative*, which specifies the action to take (like `kubectl create pod ...`, `kubectl describe deployment ...` etc), and *declarative*, which specifies objects in a yaml file and creates them with `kubectl create -f <yaml filename>`. Which to use is a matter of preference; in general, I recommend *declarative* (ie file based) commands for *any action that changes the state of the system* (ie create / update / destroy operations) so that these changes can be based off of easily tracked and version controlled config files, and *imperative* commands only for gathering information without changing anything (`kubectl get ...`, `kubectl decribe ...`). These are not strict rules (we just saw a convenient example of the imperative `kubectl create configmap` above, useful since an env-file specification is so much easier to write than the corresponding yaml), but lend themselves well to good record-keeping, automation and version control for your workloads.

4. Describe a postgres database in a pod with the following `postgres.yaml`:

```
apiVersion: v1
kind: Pod
metadata:
  name: dbdemo
  namespace: default
spec:
  containers:
  - name: pg
    image: postgres:9.6
    env:
      - name: POSTGRES_USER
        valueFrom:
          configMapKeyRef:
            name: dbconfig
            key: user
      - name: POSTGRES_DB
        valueFrom:
          configMapKeyRef:
            name: dbconfig
            key: db
```


Here we're populating the environment variables `POSTGRES_USER` and `POSTGRES_DB` from our configMap, under the `containers:env` specification. Notice that the pod definition itself makes no reference to the literal values of these environment variables; we can reconfigure our database (say for deployment in a different environment) by swapping out our `dbconfig` configMap, and leaving our pod definition untouched.

Deploy your pod as usual with `kubectl create -f postgres.yaml`.

5. Describe your `dbdemo` pod:

```
[centos@kube-0 ~]$ kubectl describe pod dbdemo

...

Environment:
  POSTGRES_USER:  <set to the key 'user' of config map 'dbconfig'>  Optional: false
  POSTGRES_DB:    <set to the key 'db' of config map 'dbconfig'>    Optional: false

...
```

You should see a block like the above, indicating that the listed environment variables are being populated from the configMap, as expected.

6. Attach to your postgres database using the username and database name you specified in `config.yaml`, to prove to yourself the configMap information actually got consumed:

```
[centos@kube-0 ~]$ kubectl exec -it -c pg dbdemo -- psql -U moby -d mydb

mydb=# \du
                                   List of roles
 Role name |                         Attributes                         | Member of
-----------+------------------------------------------------------------+-----------
 moby      | Superuser, Create role, Create DB, Replication, Bypass RLS | {}

mydb=# \q
```

The user and database got created with the names we defined.

7. Delete your pod with `kubectl delete -f postgres.yaml`.

8. So far, we've used the environment variables postgres looks for for setting user and database names. Some config, however, is expected as a file rather than an environment variable; configMaps can provision config files as well as environment config. Create a database initialization script `db-init.sh`:

```
#!/bin/bash
set -e

psql -v ON_ERROR_STOP=1 --username "$POSTGRES_USER" --dbname "$POSTGRES_DB" <<-EOSQL
    CREATE TABLE PRODUCTS(PRICE FLOAT, NAME TEXT);
    INSERT INTO PRODUCTS VALUES('18.95', 'widget');
    INSERT INTO PRODUCTS VALUES('1.45', 'sprocket');
EOSQL
```

9. Turn this entire file into a configMap:

```
[centos@kube-0 ~]$ kubectl create configmap dbinit --from-file=db-init.sh
```

10. We'll provision this config file to our postgres container by mounting it in as a volume to the correct path; postgres will run all `.sh` files found at the path `/docker-entrypoint-initdb.d` upon initialization. Change your `postgres.yaml` file to look like this:

```
apiVersion: v1
kind: Pod
metadata:
```



```
  name: dbdemo
  namespace: default
spec:
  containers:
  - name: pg
    image: postgres:9.6
    volumeMounts:
    - name: dbinit-vol
      mountPath: /docker-entrypoint-initdb.d
    env:
      - name: POSTGRES_USER
        valueFrom:
          configMapKeyRef:
            name: dbconfig
            key: user
      - name: POSTGRES_DB
        valueFrom:
          configMapKeyRef:
            name: dbconfig
            key: db
  volumes:
    - name: dbinit-vol
      configMap:
        name: dbinit
```

Here we've added the `volumeMounts` key describing which volume (`dbinit-vol`) to mount in the container and at what path (`/docker-entrypoint-initdb.d`). We've also added the `volumes` key to define the volumes themselves; we create one volume named `dbinit-vol`, populated from the files contained in the configMap `dbinit` we just created.

11. Deploy postgres with this configuration, and check that the database initialization script actually worked:

```
[centos@kube-0 ~]$ kubectl create -f postgres.yaml
[centos@kube-0 ~]$ kubectl exec -it -c pg dbdemo -- psql -U moby -d mydb

mydb=# SELECT * FROM products;

 price |   name
-------+----------
 18.95 | widget
  1.45 | sprocket
(2 rows)

mydb=# \q
```

Our table was initialized via our config file, as expected. After exiting your pod, delete it with `kubectl delete -f postgres.yaml`.

22.2 Provisioning Secrets

So far, we've provisioned non-sensitive data to our pod, but often we want options for added security when provisioning things like passwords or other access tokens. For this, Kubernetes maintains a separate config provisioning tool, *secrets*.

1. Let's set a custom password for our database using a Kubernetes secret. Create a file `secret.yaml` with the following content:

```
apiVersion: v1
kind: Secret
```



```
metadata:
  name: postgres-pwd
  namespace: default
type: Opaque
stringData:
  password: "mypassword"
```

Create the secret via `kubectl create -f secret.yaml`, This will create a secret called `postgres-pwd` that encodes our password.

> Note: Of course it's not recommended to leave your secret unencrypted in a file like `secret.yaml`; in practice, we'd delete this file as soon as the secret is created.

2. Update your `postgres.yaml` definition to look like this:

```
apiVersion: v1
kind: Pod
metadata:
  name: dbdemo
  namespace: default
spec:
  containers:
  - name: pg
    image: postgres:9.6
    volumeMounts:
    - name: dbinit-vol
      mountPath: /docker-entrypoint-initdb.d
    env:
      - name: POSTGRES_USER
        valueFrom:
          configMapKeyRef:
            name: dbconfig
            key: user
      - name: POSTGRES_DB
        valueFrom:
          configMapKeyRef:
            name: dbconfig
            key: db
      - name: POSTGRES_PASSWORD
        valueFrom:
          secretKeyRef:
            name: postgres-pwd
            key: password
  volumes:
    - name: dbinit-vol
      configMap:
        name: dbinit
```

This is exactly the same as above, but adds the block at the bottom which populates the `POSTGRES_PASSWORD` environment variable in the pg container with the value found under the `password` key in the `postgres-pwd` secret.

3. Create your pod, and dump its postgres environment variables:

```
[centos@kube-0 ~]$ kubectl create -f postgres.yaml
[centos@kube-0 ~]$ kubectl exec -it -c pg dbdemo -- env | grep POSTGRES

POSTGRES_USER=moby
POSTGRES_DB=mydb
```



```
POSTGRES_PASSWORD=mypassword
```

The `POSTGRES_PASSWORD` has been provisioned from your Kube secret.

4. Note that anyone with `kubectl get secret` permissions can recover your secret as follows:

```
[centos@kube-0 ~]$ kubectl get secret postgres-pwd -o yaml

apiVersion: v1
data:
  password: bXlwYXNzd29yZA==
kind: Secret
metadata:
  creationTimestamp: "2019-01-07T18:42:01Z"
  name: postgres-pwd
  namespace: default
  resourceVersion: "604412"
  selfLink: /api/v1/namespaces/default/secrets/postgres-pwd
  uid: ebb6f645-12ab-11e9-87ee-0242ac11000a
type: Opaque

[centos@kube-0 ~]$ echo 'bXlwYXNzd29yZA==' | base64 --decode

mypassword
```

Challenge: the secret password can also be recovered in plain text on the node hosting the postgres pod. Can you find it?

5. As usual, clean up by deleting the pods, configMaps, and secret you created in this exercise.

22.3 Conclusion

In this exercise, we looked at configMaps and secrets, two tools for provisioning information to your deployments. When deciding where to place configuration, it can help to prioritize designing for reusability; in the postgres example we saw, we separated out all the environment config from the pod definition, so the same pod yaml could be migrated from environment to environment with no changes; all the environment specific data was captured in the configMaps and secret. If you find yourself having to do heavy reconfiguration of your pods and deployments (or even images) as you migrate from one environment to another, consider if it would be possible to separate this configuration from your pod definition using a configMap or secret.

23 Kubernetes Networking

By the end of this exercise, you should be able to:

- Predict what routing tables rules calico will write to each host in your cluster
- Route and load balance traffic to deployments using clusterIP and nodePort services
- Reconfigure a deployment into a daemonSet (analogous to changing scheduling from 'replicated' to 'global' in a swarm service)

23.1 Routing Traffic with Calico

1. Make sure you're on the master node `kube-0`, and redeploy the nginx deployment defined in `deployment.yaml` from the last exercise.
2. List your pods:


```
[centos@kube-0 ~]$ kubectl get pods
```

3. Get some metadata on one of the pods found in the last step:

```
[centos@kube-0 ~]$ kubectl describe pods <pod name>
```

which in my case results in:

```
Name:               nginx-deployment-69df458bc5-cg4mk
Namespace:          default
Priority:           0
PriorityClassName:  <none>
Node:               kube-1/10.10.95.205
Start Time:         Tue, 21 Aug 2018 14:47:48 +0000
Labels:             app=nginx
                    pod-template-hash=2589014671
Annotations:        <none>
Status:             Running
IP:                 192.168.126.80
Controlled By:      ReplicaSet/nginx-deployment-69df458bc5
Containers:
  nginx:
    Container ID:   docker://664616e...
    Image:          nginx:1.7.9
    Image ID:       docker-pullable://nginx@sha256:e3456c8...
    Port:           <none>
    Host Port:      <none>
    State:          Running
      Started:      Tue, 21 Aug 2018 14:47:50 +0000
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-5ggnn (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Volumes:
  default-token-5ggnn:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-5ggnn
    Optional:    false
QoS Class:       BestEffort
Node-Selectors:  <none>
Tolerations:     node.kubernetes.io/not-ready:NoExecute for 300s
                 node.kubernetes.io/unreachable:NoExecute for 300s
Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  1m    default-scheduler  Successfully assigned default/
                                              nginx-deployment-69df458bc5-cg4mk
                                              to kube-1
  Normal  Pulled     1m    kubelet, kube-1    Container image "nginx:1.7.9"
                                              already present on machine
```



```
  Normal  Created    1m    kubelet, kube-1    Created container
  Normal  Started    1m    kubelet, kube-1    Started container
```

We can see that in our case the pod has been deployed to `kube-1` as indicated near the top of the output, and the pod has an IP of `192.168.126.80`.

4. Have a look at the routing table on `kube-0` using `ip route`, which for my example looks like:

```
[centos@kube-0 ~]$ ip route

default via 10.10.64.1 dev eth0
10.10.64.0/20 dev eth0  proto kernel  scope link  src 10.10.68.222
172.17.0.0/16 dev docker0  proto kernel  scope link  src 172.17.0.1
192.168.126.64/26 via 10.10.95.205 dev tunl0  proto bird onlink
blackhole 192.168.145.64/26  proto bird
192.168.145.65 dev cali6edd91665ff  scope link
192.168.145.66 dev calia9ee759be59  scope link
```

Notice the fourth line; this rule was written by Calico to send any traffic on the 192.168.126.64/26 subnet (which the pod we examined above is on) to the host at IP 10.10.95.205 via IP in IP as indicated by the `dev tunl0` entry. Look at your own routing table and list of VM IPs; what are the corresponding subnets, pod IPs and host IPs in your case? Does that make sense based on the host you found for the nginx pod above?

5. Curl your pod's IP on port 80 from `kube-0`; you should see the HTML for the nginx landing page. By default this pod is reachable at this IP from anywhere in the Kubernetes cluster.

6. Head over to the node this pod got scheduled on (`kube-1` in the example above), and have a look at that host's routing table in the same way:

```
[centos@kube-1 ~]$ ip route

default via 10.10.80.1 dev eth0
10.10.80.0/20 dev eth0  proto kernel  scope link  src 10.10.95.205
172.17.0.0/16 dev docker0  proto kernel  scope link  src 172.17.0.1
blackhole 192.168.126.64/26  proto bird
192.168.126.78 dev calib0aa6a43271  scope link
192.168.126.79 dev cali3489915b309  scope link
192.168.126.80 dev cali2d894f7a3f6  scope link
192.168.145.64/26 via 10.10.68.222 dev tunl0  proto bird onlink
```

Again notice the second-to-last line; this time, the pod IP is routed to a `cali***` device, which is a virtual ethernet endpoint in the host's network namespace, providing a point of ingress into that pod. Once again try `curl <pod IP>:80` - you'll see the nginx landing page html as before.

7. Back on `kube-0`, fetch the logs generated by the pod you've been curling:

```
[centos@kube-0 ~]$ kubectl logs <pod name>

10.10.52.135 - - [09/May/2018:13:58:42 +0000]
    "GET / HTTP/1.1" 200 612 "-" "curl/7.29.0" "-"
192.168.84.128 - - [09/May/2018:14:00:41 +0000]
    "GET / HTTP/1.1" 200 612 "-" "curl/7.29.0" "-"
```

We see records of the curls we preformed above; like Docker containers, these logs are the STDOUT and STDERR of the containerized processes.

23.2 Routing and Load Balancing with Services

1. Above we were able to hit nginx at the pod IP, but there is no guarantee this pod won't get rescheduled to a new IP. If we want a stable IP for this deployment, we need to create a `ClusterIP` service. In a file `cluster.yaml`

on your master kube-0:

```
apiVersion: v1
kind: Service
metadata:
  name: cluster-demo
spec:
  selector:
    app: nginx
  ports:
  - port: 8080
    targetPort: 80
```

Create this service with `kubectl create -f cluster.yaml`. This maps the pod internal port 80 to the cluster wide external port 8080; furthermore, this IP and port will only be reachable from *within* the cluster. Also note the `selector: app: nginx` specification; that indicates that this service will route traffic to every pod that has `nginx` as the value of the app label in this namespace.

2. Let's see what services we have now:

```
[centos@kube-0 ~]$ kubectl get services
NAME         TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)    AGE
kubernetes     ClusterIP   10.96.0.1       <none>        443/TCP    33m
cluster-demo   ClusterIP   10.104.201.93   <none>        8080/TCP   48s
```

The second one is the one we just created and we can see that a stable IP address and port `10.104.201.93:8080` has been assigned to our `nginx` service.

3. Let's try to access Nginx now, from any node in our cluster:

```
[centos@kube-0 ~]$ curl <nginx CLUSTER-IP>:8080
```

which should return the Nginx welcome page. Even if pods get rescheduled to new IPs, this clusterIP service will preserve a stable entrypoint for traffic to be load balanced across all pods matching the service's label selector.

4. ClusterIP services are reachable only from within the Kubernetes cluster. If you want to route traffic to your pods from an external network, you'll need a NodePort service. On your master kube-0, create a file `nodeport.yaml`:

```
apiVersion: v1
kind: Service
metadata:
  name: nodeport-demo
spec:
  type: NodePort
  selector:
      app: nginx
  ports:
  - port: 8080
    targetPort: 80
```

Create this service with `kubectl create -f nodeport.yaml`. Notice this is exactly the same as the ClusterIP service definition, but now we're requesting a type NodePort.

5. Inspect this service's metadata:

```
[centos@kube-0 ~]$ kubectl describe service nodeport-demo
```

Notice the NodePort field: this is a randomly selected port from the range 30000-32767 where your pods will be reachable externally. Try visiting your nginx deployment at any public IP of your cluster, and the port you found above, and confirming you can see the nginx landing page.

6. Clean up the objects you created in this section:

```
[centos@kube-0 ~]$ kubectl delete deployment nginx-deployment
[centos@kube-0 ~]$ kubectl delete service cluster-demo
[centos@kube-0 ~]$ kubectl delete service nodeport-demo
```

23.3 Optional: Deploying DockerCoins onto the Kubernetes Cluster

1. First deploy Redis via `kubectl create deployment`:

```
[centos@kube-0 ~]$ kubectl create deployment redis --image=redis
```

2. And now all the other deployments. To avoid too much typing we do that in a loop:

```
[centos@kube-0 ~]$ for DEPLOYMENT in hasher rng webui worker; do
    kubectl create deployment $DEPLOYMENT \
        --image=training/dockercoins-${DEPLOYMENT}:1.0
done
```

3. Let's see what we have:

```
[centos@kube-0 ~]$ kubectl get pods -o wide -w
```

in my case the result is:

```
hasher-6c64f78655-rgjk5   1/1       Running   0          53s       10.36.0.1   kube-1
redis-75586d7d7c-mmjg7    1/1       Running   0          5m        10.44.0.2   kube-1
rng-d94d56d4f-twlwz       1/1       Running   0          53s       10.44.0.1   kube-1
webui-6d8668984d-sqtt8    1/1       Running   0          52s       10.36.0.2   kube-1
worker-56756ddbb8-lbv9r   1/1       Running   0          52s       10.44.0.3   kube-1
```

pods have been distributed across our cluster.

4. We can also look at some logs:

```
[centos@kube-0 ~]$ kubectl logs deploy/rng
[centos@kube-0 ~]$ kubectl logs deploy/worker
```

The `rng` service (and also the `hasher` and `webui` services) seem to work fine but the `worker` service reports errors. The reason is that unlike on Swarm, Kubernetes does not automatically provide a stable networking endpoint for deployments. We need to create at least a `ClusterIP` service for each of our deployments so they can communicate.

5. List your current services:

```
[centos@kube-0 ~]$ kubectl get services
NAME         TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   46m
```

6. Expose the `redis`, `rng` and `hasher` internally to your cluster, specifying the correct internal port:

```
[centos@kube-0 ~]$ kubectl expose deployment redis --port 6379
[centos@kube-0 ~]$ kubectl expose deployment rng --port 80
[centos@kube-0 ~]$ kubectl expose deployment hasher --port 80
```

7. List your services again:

```
[centos@kube-0 ~]$ kubectl get services
NAME         TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
hasher       ClusterIP   10.108.207.22    <none>        80/TCP     20s
kubernetes   ClusterIP   10.96.0.1        <none>        443/TCP    47m
redis        ClusterIP   10.100.14.121    <none>        6379/TCP   31s
rng          ClusterIP   10.111.235.252   <none>        80/TCP     26s
```


Evidently `kubectl expose` creates `ClusterIP` services allowing stable, internal reachability for your deployments, much like you did via yaml manifests for your nginx deployment in the last section. See the `kubectl` api docs for more command-line alternatives to yaml manifests.

8. Get the logs of the worker again:

```
[centos@kube-0 ~]$ kubectl logs deploy/worker
```

This time you should see that the `worker` recovered (give it at least 10 seconds to do so). The `worker` can now access the other services.

9. Now let's expose the `webui` to the public using a service of type `NodePort`:

```
[centos@kube-0 ~]$ kubectl expose deploy/webui --type=NodePort --port 80
```

10. List your services one more time:

```
[centos@kube-0 ~]$ kubectl get services
NAME         TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
hasher       ClusterIP   10.108.207.22    <none>        80/TCP         2m
kubernetes   ClusterIP   10.96.0.1        <none>        443/TCP        49m
redis        ClusterIP   10.100.14.121    <none>        6379/TCP       2m
rng          ClusterIP   10.111.235.252   <none>        80/TCP         2m
webui        NodePort    10.108.88.182    <none>        80:32015/TCP   33s
```

Notice the `NodePort` service created for `webui`. This type of service provides similar behavior to the Swarm L4 mesh net: a port (32015 in my case) has been reserved across the cluster; any external traffic hitting any cluster IP on that port will be directed to port 80 inside a `webui` pod.

11. Visit your Dockercoins web ui at `http://<node IP>:<port>`, where `<node IP>` is the public IP address any of your cluster members. You should see the dashboard of our DockerCoins application.

12. Let's scale up the worker a bit and see the effect of it:

```
[centos@kube-0 ~]$ kubectl scale deploy/worker --replicas=10
```

Observe the result of this scaling in the browser. We do not really get a 10-fold increase in throughput, just as when we deployed DockerCoins on swarm; the `rng` service is causing a bottleneck.

13. To scale up, we want to run an instance of `rng` on each node of the cluster. For this we use a `DaemonSet`. We do this by using a yaml file that captures the desired configuration, rather than through the CLI.

Create a file `deploy-rng.yaml` as follows:

```
[centos@kube-0 ~]$ kubectl get deploy/rng -o yaml --export > deploy-rng.yaml
```

Note: `--export` will remove "cluster-specific" information

14. Edit this file to make it describe a `DaemonSet` instead of a `Deployment`:

- change `kind` to `DaemonSet`
- remove the `progressDeadlineSeconds` field
- remove the `replicas` field
- remove the `strategy` block (which defines the rollout mechanism for a deployment)
- remove the `status: {}` line at the end

15. Now apply this YAML file to create the `DaemonSet`:

```
[centos@kube-0 ~]$ kubectl apply -f deploy-rng.yaml
```

16. We can now look at the `DaemonSet` that was created:

```
[centos@kube-0 ~]$ kubectl get daemonset

NAME      DESIRED   CURRENT   READY     UP-TO-DATE   AVAILABLE   NODE SELECTOR   AGE
rng       1         1         1         1            1           <none>          1m
```


Since we only have one workload-bearing node in our cluster (`kube-1`), we still only get one rng pod and no performance improvement - but, if we added more non-master nodes to our Kubernetes cluster, they would get random number generator pods scheduled on them as they joined, relaxing the bottleneck.

17. If we do a `kubectl get all` we will see that we now have both a `deployment.apps/rng` AND a `daemonset.apps/rng`. Deployments are not just converted to DaemonSets. Let's delete the `rng` deployment:

```
[centos@kube-0 ~]$ kubectl delete deploy/rng
```

18. Clean up your resources when done:

```
[centos@kube-0 ~]$ for D in redis hasher rng webui; \
    do kubectl delete svc/$D; done
[centos@kube-0 ~]$ for D in redis hasher webui worker; \
    do kubectl delete deploy/$D; done
[centos@kube-0 ~]$ kubectl delete ds/rng
```

19. Make sure that everything is cleared:

```
[centos@kube-0 ~]$ kubectl get all
```

should only show the `svc/kubernetes` resource.

23.4 Conclusion

In this exercise, we looked at some of the key Kubernetes service objects that provide routing and load balancing for collections of pods; clusterIP for internal communication, analogous to Swarm's VIPs, and NodePort, for routing external traffic to an app similarly to Swarm's L4 mesh net. We also briefly touched on the inner workings of Calico, one of many Kubernetes network plugins and the one that ships natively with Docker's Enterprise Edition product. The key networking difference between Swarm and Kubernetes is their approach to default firewalling; while Swarm firewalls software defined networks automatically, all pods can reach all other pods on a Kube cluster, in Calico's case via the BGP-updated control plane and IP-in-IP data plane you explored above.

24 Cleaning up Docker Resources

By the end of this exercise, you should be able to:

- Assess how much disk space docker objects are consuming
- Use `docker prune` commands to clear out unneeded docker objects
- Apply label based filters to `prune` commands to control what gets deleted in a cleanup operation

1. Find out how much memory Docker is using by executing:

   ```
   PS: node-0 Administrator> 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:

   ```
   PS: node-0 Administrator> 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?):

   ```
   PS: node-0 Administrator> docker container run `
       --label apple --name fuji -d `
       mcr.microsoft.com/windows/nanoserver:10.0.17763.737

   PS: node-0 Administrator> docker container run `
   ```



```
    --label orange --name clementine -d `
    mcr.microsoft.com/windows/nanoserver:10.0.17763.737
```

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

```
PS: node-0 Administrator> docker container ls -a
PS: node-0 Administrator> docker container prune --filter 'label=apple'
PS: node-0 Administrator> 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:

```
PS: node-0 Administrator> $dt=date
PS: node-0 Administrator> $until='until=' + $dt.ToString("yyyy-MM-dd'T'HH:mm:ss.fffK")
PS: node-0 Administrator> docker container run `
    --label tomato --name beefsteak -d `
    mcr.microsoft.com/windows/nanoserver:10.0.17763.737
```

   And use the timestamp returned in a prune:

```
PS: node-0 Administrator> docker container prune -f --filter $until
PS: node-0 Administrator> 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.

6. Cleanup all containers to end the exercise:

```
PS: node-0 Administrator> docker container rm $(docker container ls -aq)
```

24.1 Conclusion

In this exercise, we saw some very basic `docker prune` usage - most of the top-level docker objects have a `prune` command (`docker container prune`, `docker volume prune` etc). Most docker objects leave something on disk even after being shut down; consider using these cleanup commands as part of your cluster maintenance and garbage collection plan, to avoid accidentally running out of disk on your Docker hosts.

25 Inspection Commands

By the end of this exercise, you should be able to:

- Gather system level info from the docker engine
- Consume and format the docker engine's event stream for monitoring purposes

25.1 Inspecting System Information

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

```
PS: node-0 Administrator> 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 kernel version of Windows are you running?
   - whether Docker is running in swarm mode?

25.2 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:

   ```
   PS: node-0 Administrator> 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 powershell window and execute the following command:

   ```
   PS: node-0 Administrator> docker container run --rm `
          mcr.microsoft.com/windows/nanoserver:10.0.17763.737 `
          ping 8.8.8.8
   ```

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

   ```
   2019-10-31T14:57:42.365968600Z container create fa55...
   2019-10-31T14:57:42.372964200Z container attach fa55...
   2019-10-31T14:57:42.467961900Z network connect a7c2...
   2019-10-31T14:57:42.931061900Z container start fa55...
   2019-10-31T14:57:46.237482500Z container die fa55...
   2019-10-31T14:57:46.262481400Z network disconnect a7c2...
   2019-10-31T14:57:46.335474200Z container destroy fa55...
   ```

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:

   ```
   PS: node-0 Administrator> docker system events --format '--> {{.Type}}-{{.Action}}'
   ```

   now the output looks a little bit less cluttered when we rerun our nanoserver 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):

   ```
   PS: node-0 Administrator> docker events --format '{{json .}}'
   ```

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

   ```
   {
   "status": "create",
   "id": "3a8d28972026945ca27727e48bcbc66ae7539ecbe0a85d3c3d82d4c34463954f",
   "from": "mcr.microsoft.com/windows/nanoserver:10.0.14393.2551",
   "Type": "container",
   "Action": "create",
   "Actor": {
       "ID": "3a8d28972026945ca27727e48bcbc66ae7539ecbe0a85d3c3d82d4c34463954f",
       "Attributes": {
       "image": "mcr.microsoft.com/windows/nanoserver:10.0.14393.2551",
       "name": "festive_engelbart"
       }
   },
   "scope": "local",
   "time": 1502850178,
   "timeNano": 1502850178980991200
   }
   ```


25.3 Conclusion

In this exercise we have learned how to inspect system wide properties of our Docker host by using the `docker system info` command; this is one of the first places to look for general config information to include in a bug report. We also saw a simple example of `docker system events`; the events stream is one of the primary sources of information that should be logged and monitored when running Docker in production. Many commercial as well as open source products (such as Elastic Stack) exist to facilitate aggregating and mining these streams at scale.

26 Starting a Compose App

By the end of this exercise, you should be able to:

- Read a basic docker compose yaml file and understand what components it is declaring
- Start, stop, and inspect the logs of an application defined by a docker compose file

26.1 Preparing Service Images

1. Download the Dockercoins app from github:

   ```
   PS: node-0 Administrator> git clone -b ee3.0-ws19 `
       https://github.com/docker-training/orchestration-workshop-net.git
   PS: node-0 Administrator> cd orchestration-workshop-net
   ```

   This app consists of 5 services: a random number generator `rng`, a `hasher`, a backend `worker`, a `redis` queue, and a `web` frontend; the code you just downloaded has the source code for each process and a Dockerfile to containerize each of them.

2. Have a brief look at the source for each component of your application. Each folder under `~/orchestration-workshop-net/` contains the application logic for the component, and a Dockerfile for building that logic into a Docker image. We've pre-built these images as `training/dc_rng:1.0`, `training/dc_worker:1.0` et cetera, so no need to build them yourself.

3. Have a look in `docker-compose.yml`; especially notice the `services` section. Each block here defines a different Docker service. They each have exactly one image which containers for this service will be started from, as well as other configuration details like network connections and port exposures. Full syntax for Docker Compose files can be found here: https://dockr.ly/2iHUpeX.

26.2 Starting the App

1. Stand up the app:

   ```
   PS: node-0 orchestration-workshop-net> docker-compose up
   ```

   After a moment, your app should be running; visit `<node 0 public IP>:8000` to see the web frontend visualizing your rate of Dockercoin mining.

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`, and start the app again in the background:

   ```
   PS: node-0 orchestration-workshop-net> docker-compose up -d
   ```

3. Check out which containers are running thanks to Compose (`Names` column truncated for clarity):

   ```
   PS: node-0 orchestration-workshop-net> docker-compose ps

   Name       Command                          State   Ports
   ------------------------------------------------------------------------
   hasher_1   dotnet run                       Up      0.0.0.0:8002->80/tcp
   redis_1    redis-server.exe C:\Redis\ ...   Up      6379/tcp
   rng_1      dotnet run                       Up      0.0.0.0:8001->80/tcp
   ```



```
webui_1    node webui.js                    Up      0.0.0.0:8000->80/tcp
worker_1   dotnet run                       Up
```

4. Compare this to the usual `docker container ls`; do you notice any differences? If not, start a couple of extra containers using `docker container run...`, and check again.
5. See logs from a Compose-managed app via:

```
PS: node-0 orchestration-workshop-net> docker-compose logs
```

26.3 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; all runtime configuration for each container 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.

27 Scaling a Compose App

By the end of this exercise, you should be able to:

- Scale a service from Docker Compose up or down

27.1 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 by redeploying the app with the `--scale` flag, while checking the list of running containers before and after:

```
PS: node-0 orchestration-workshop-net> docker-compose ps
PS: node-0 orchestration-workshop-net> docker-compose up -d --scale worker=2
PS: node-0 orchestration-workshop-net> 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.

27.2 Investigating Bottlenecks

1. Open the task manager:

```
PS: node-0 orchestration-workshop-net> taskmgr
```

   CPU and memory usage should be fairly negligible. So, keep scaling up your workers to mine more dockercoins:

```
PS: node-0 orchestration-workshop-net> docker-compose up -d --scale worker=10
PS: node-0 orchestration-workshop-net> 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.


```
PS: node-0 orchestration-workshop-net> httping -c 10 <node 0 public IP>:8001
PS: node-0 orchestration-workshop-net> httping -c 10 <node 0 public IP>:8002
```

rng 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.

4. For now, shut your app down:

```
PS: node-0 orchestration-workshop-net> docker-compose down
```

27.3 Conclusion

In this exercise, we saw how to scale up a service defined in our Compose app using the `--scale` flag. 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.

Instructor Demos

1 Instructor Demo: Process Isolation

In this demo, we'll illustrate:

- What containerized process IDs look like inside versus outside of a namespace
- How to impose resource limitations on CPU and memory consumption of a containerized process

1.1 Exploring the PID Namespace

1. Start a simple container we can explore:

```
PS: node-0 Administrator> docker container run -d --name pinger `
    mcr.microsoft.com/powershell:preview-windowsservercore-1809 ping -t 8.8.8.8
```

2. Launch a child process inside this container to display all the processes running inside it:

```
PS: node-0 Administrator> docker container exec pinger powershell Get-Process

Handles  NPM(K)    PM(K)      WS(K)     CPU(s)      Id  SI ProcessName
-------  ------    -----      -----     ------      --  -- -----------
    120       6     1220       4796       0.02    432   3 CExecSvc
     79       5      920       3632       0.00   6244   3 CompatTelRunner
    151      10     6836      12528       0.02   5408   3 conhost
    222      11     2072       4988       0.11   7076   3 csrss
     49       6     1004       3276       0.02   5540   3 fontdrvhost
      0       0       56          8                 0   0 Idle
    777      22     4340      12820       0.11   7276   3 lsass
     68       6      884       3240       0.02   7452   3 PING
    506      34    66768      73244       3.23   6824   3 powershell
    219      12     2796       6272       0.14   6480   3 services
     50       4      536       1208       0.11    568   0 smss
    767      28     6540      18568       0.20   1496   3 svchost
    396      16     7560      13700       0.09   3376   3 svchost
    485      21     7928      19780       0.27   3668   3 svchost
    188      15     3052       8880       0.06   4272   3 svchost
    348      14     2896       9848       0.06   4588   3 svchost
    144       9     1764       6364       0.05   5832   3 svchost
    123       7     1344       5716       0.02   6784   3 svchost
```



```
   482      35     5692       17820       1.81   7004   3 svchost
   309      16     2592        8132       0.08   7620   3 svchost
  2981       0      196         152      39.72      4   0 System
   188      12     2020        7136       0.05   6208   3 wininit
```

In Windows containers, a whole set of system processes need to run in order for the intended application process to be executed successfully. Just like a regular Windows process list, we see the root `Idle` process at PID 0, and the `System` process at PID 4.

Another way to achieve a similar result is to use `container top`:

```
PS: node-0 Administrator> docker container top pinger
```

```
Name                PID                 CPU                 Private Working Set
smss.exe            568                 00:00:00.109        286.7kB
csrss.exe           7076                00:00:00.109        1.106MB
wininit.exe         6208                00:00:00.046        1.204MB
services.exe        6480                00:00:00.140        1.602MB
lsass.exe           7276                00:00:00.109        3.453MB
svchost.exe         4588                00:00:00.062        2.109MB
fontdrvhost.exe     5540                00:00:00.015        548.9kB
svchost.exe         7620                00:00:00.078        2.015MB
svchost.exe         1496                00:00:00.218        5.173MB
svchost.exe         4272                00:00:00.062        2.417MB
CExecSvc.exe        432                 00:00:00.031        860.2kB
svchost.exe         3376                00:00:00.093        5.62MB
PING.EXE            7452                00:00:00.015        548.9kB
svchost.exe         7004                00:00:01.812        4.092MB
svchost.exe         6784                00:00:00.015        876.5kB
svchost.exe         3668                00:00:00.265        6.513MB
svchost.exe         5832                00:00:00.046        1.11MB
CompatTelRunner.exe 6244                00:00:00.000        589.8kB
conhost.exe         5408                00:00:00.015        6.304MB
```

3. Run `Get-Process` directly on your host. The ping process is visible there, but so are all the other processes on this machine; the container's namespaces isolated what `Get-Process` returns when executed as a child process within the container.

4. List your containers to show that the `pinger` container is still running:

   ```
   PS: node-0 Administrator> docker container ls
   ```

   Kill the ping process by host PID, confirm with `Y` to stop the process, and show the container has stopped:

   ```
   PS: node-0 Administrator>Stop-Process -Id [PID of ping]
   PS: node-0 Administrator>docker container ls

   CONTAINER ID   IMAGE   COMMAND   CREATED   STATUS   PORTS   NAMES
   ```

   Killing the ping process on the host also kills the container. Note using `Stop-Process` is just for demonstration purposes here; never stop containers this way.

1.2 Imposing Resource Limitations

1. Open the Task Manager, either through the search bar or by typing `taskmgr` in the command prompt. Then click **More Details** in the task manager to get a live report of resource consumption.

2. Start a container designed to simulate cpu and memory load:

   ```
   PS: node-0 Administrator> docker container run -it training/winstress:ws19 pwsh.exe
   ```

3. Execute a script inside your container to allocate memory as fast as possible:


```
PS C:\> .\saturate-mem.ps1
```

You should see the `Memory` column on the Task Manager increase quickly, even turning red after a while. Then, this error message should be thrown (CTRL+c to break the loop):

```
Exception of type 'System.OutOfMemoryException' was thrown.
At C:\saturate-mem.ps1:2 char:37
+ ...  -lt 100000; $i++) { $mem_stress += ("a" * 1023MB) + ("b" * 1023MB) }
+                          ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : OperationStopped: (:) [], OutOfMemoryException
    + FullyQualifiedErrorId : System.OutOfMemoryException
```

Note this may even disrupt your RDP connection to your VM - failing to constrain resource consumption can be catastrophic.

4. `CTRL+C` to kill this memory-saturating process. Then, exit and remove the container to release the allocated memory:

   ```
   PS: node-0 Administrator>docker container rm -f <container ID>
   ```

   Immediately, the memory in the Task Manager should drop.

5. Now, let's start a container with a memory limit:

   ```
   PS: node-0 Administrator> docker container run `
       -it -m 4096mb training/winstress:ws19 pwsh.exe
   ```

6. Run the same script to generate memory pressure:

   ```
   PS C:\> .\saturate-mem.ps1
   ```

   While the memory does increase in the Task Manager, allocations get cut off before the system memory is completely consumed. `CTRL+C` to kill the process, and exit the container again.

7. Remove this container.

   ```
   PS: node-0 Administrator>docker container rm -f <container ID>
   ```

1.3 Conclusion

In this demo, we explored some of the most important technologies that make containerization possible: namespaces and control groups. The core message here is that containerized processes are just processes running on their host, isolated and constrained by these technologies. All the tools and management strategies you would use for conventional processes apply just as well for containerized processes.

2 Instructor Demo: Creating Images

In this demo, we'll illustrate:

- How to read each step of the image build output
- How intermediate image layers behave in the cache and as independent images
- What the meanings of 'dangling' and `<missing>` image layers are

2.1 Understanding Image Build Output

1. Make a folder `demo` for our image demo:

   ```
   PS: node-0 Administrator> mkdir demo ; cd demo
   ```

   In this folder, create a `Dockerfile`:


```
FROM mcr.microsoft.com/windows/servercore:10.0.17763.805
SHELL ["powershell", "-Command"]
RUN iex (invoke-webrequest https://chocolatey.org/install.ps1 -UseBasicParsing)
RUN choco install -y which
RUN choco install -y wget
RUN choco install -y vim
```

2. Build the image from the `Dockerfile`:

   ```
   PS: node-0 demo> docker image build -t demo .
   ```

3. Examine the output from the build process. The very first line looks like:

   ```
   Sending build context to Docker daemon  2.048kB
   ```

   Here the Docker daemon is archiving everything at the path specified in the `docker image build` command (`.` or the current directory in this example). This is why we made a fresh directory `demo` to build in, so that nothing extra is included in this process.

4. The next two lines look like this:

   ```
   Step 1/6 : FROM mcr.microsoft.com/windows/servercore:10.0.17763.805
    ---> 42277f7f55c3
   ```

   Do a `docker image ls`:

   ```
   REPOSITORY                     TAG      IMAGE ID       CREATED        SIZE
   demo                           latest   889ea81f9564   13 hours ago   11GB
   servercore:10.0.17763.805      latest   42277f7f55c3   6 days ago     4.79GB
   ```

   Notice the Image ID for `mcr.microsoft.com/windows/servercore:10.0.17763.805` matches that second line in the build output. The build starts from the base image defined in the `FROM` command.

5. The next few lines look like:

   ```
   Step 2/6 : SHELL powershell -Command
   ---> Running in c84a289effac
   ```

   This is the output of the `SHELL` command, `powershell -Command`. The line `Running in c84a289effac` specifies a container that this command is running in, which is spun up based on all previous image layers (just the `mcr.microsoft.com/windows/servercore:10.0.17763.805` base at the moment). Scroll down a bit and you should see something like:

   ```
   ---> e573fdd8f035
   Removing intermediate container c84a289effac
   ```

   At the end of this first `SHELL` command, the temporary container `c84a289effac` is saved as an image layer `e573fdd8f035`, and the container is removed. This is the exact same process as when you used `docker container commit` to save a container as a new image layer, but now running automatically as part of a Dockerfile build.

6. Look at the history of your image:

   ```
   docker image history demo
   ```

   ```
   IMAGE          CREATED         CREATED BY                                      SIZE
   10598cb26c8e   2 minutes ago   powershell choco install -y vim                 141MB
   4a746a1f589f   3 minutes ago   powershell choco install -y wget                31.9MB
   e6af1aae39ef   3 minutes ago   powershell choco install -y which               6.7MB
   1f167ec4ff77   2 weeks ago     powershell iex (invoke-webrequest h…            59.6MB
   e883ed43fca2   2 weeks ago     powershell #(nop)  SHELL [powershel…            41kB
   42277f7f55c3   3 weeks ago     Install update 10.0.17763.805-amd64             1.32GB
   <missing>      13 months ago   Apply image 10.0.17763.1-amd64                  3.47GB
   ```

   As you can see, the different layers of `demo` correspond to a separate line in the Dockerfile and the layers have their own ID. You can see the image layer `e573fdd8f035` committed in the second build step in the list of layers for this image.

7. Look through your build output for where steps 3/6 (installing chocolatey), 4/6 (installing which), 5/6 (installing wget), and 6/6 (installing vim) occur - the same behavior of starting a temporary container based on the previous image layers, running the RUN command, saving the container as a new image layer visible in your docker iamge history output, and deleting the temporary container is visible.

8. Every layer can be used as you would use any image, which means we can inspect a single layer. Let's inspect the `wget` layer, which in my case is `4a746a1f589f` (yours will be different, look at your `docker image history` output):

   ```
   PS: node-0 demo> docker image inspect <layer ID>
   ```

9. Let's look for the command associated with this image layer by using `--format`:

   ```
   PS: node-0 demo> docker image inspect --format='{{.ContainerConfig.Cmd}}' <layer ID>

   [powershell -Command choco install -y wget]
   ```

10. We can even start containers based on intermediate image layers; start an interactive container based on the `wget` layer, and look for whether `wget` and `vim` are installed:

    ```
    PS: node-0 demo> docker container run -it <layer ID> powershell
    PS C:\> which wget
    C:\ProgramData\chocolatey\bin\wget.exe

    PS C:\> which vim
    Not found
    ```

    `wget` is installed in this layer, but since `vim` didn't arrive until the next layer, it's not available here.

2.2 Managing Image Layers

1. Change the last line in the `Dockerfile` from the last section to install `nano` instead of `vim`:

   ```
   FROM mcr.microsoft.com/windows/servercore:10.0.17763.805
   SHELL ["powershell", "-Command"]
   RUN iex (invoke-webrequest https://chocolatey.org/install.ps1 -UseBasicParsing)
   RUN choco install -y which
   RUN choco install -y wget
   RUN choco install -y nano
   ```

2. Rebuild your image, and list your images again:

   ```
   PS: node-0 demo> docker image build -t demo .
   PS: node-0 demo> docker image ls

   REPOSITORY  TAG             IMAGE ID       CREATED         SIZE
   demo        latest          d405556fb8ad   6 seconds ago   4.91GB
   <none>      <none>          10598cb26c8e   7 minutes ago   5.03GB
   servercore  10.0.17763.805  42277f7f55c3   3 weeks ago     4.79GB
   ```

   What is that image named `<none>`? Notice the image ID is the same as the old image ID for `demo:latest` (see your history output above). The name and tag of an image is just a pointer to the stack of layers that make it up; reuse a name and tag, and you are effectively moving that pointer to a new stack of layers, leaving the old one (the one containing the `vim` install in this case) as an untagged or 'dangling' image.

3. Rewrite your `Dockerfile` one more time, to combine some of those install steps:

   ```
   FROM mcr.microsoft.com/windows/servercore:10.0.17763.805
   SHELL ["powershell", "-Command"]
   RUN iex (invoke-webrequest https://chocolatey.org/install.ps1 -UseBasicParsing)
   RUN choco install -y which wget nano
   ```


Rebuild using a `new` tag this time, and use `docker image inspect` to pull out the size of both this and your previous image, tagged `latest`:

```
PS: node-0 demo> docker image build -t demo:new .

PS: node-0 demo> docker image inspect --format '{{json .Size}}' demo:latest
4906923551
PS: node-0 demo> docker image inspect --format '{{json .Size}}' demo:new
4892480752
```

Image `demo:new` is smaller in size than `demo:latest`, even though it contains the exact same software - why?

2.3 Conclusion

In this demo, we explored the layered structure of images; each layer is built as a distinct image and can be treated as such, on the host where it was built. This information is preserved on the build host for use in the build cache; build another image based on the same lower layers, and they will be reused to speed up the build process. Notice that the same is not true of downloaded images like `mcr.microsoft.com/windows/servercore:10.0.17763.805`; intermediate image caches are not downloaded, but rather only the final complete image.

3 Instructor Demo: Basic Volume Usage

In this demo, we'll illustrate:

- Creating, updating, destroying, and mounting docker named volumes
- How volumes interact with a container's layered filesystem
- Usecases for mounting host directories into a container

3.1 Using Named Volumes

1. Create a volume, and inspect its metadata:

```
PS: node-0 Administrator> docker volume create demovol
PS: node-0 Administrator> docker volume inspect demovol

[
    {
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "C:\\ProgramData\\docker\\volumes\\demovol\\_data",
        "Name": "demovol",
        "Options": {},
        "Scope": "local"
    }
]
```

   We can see that by default, named volumes are created under `C:\\ProgramData\\docker\\volumes\\<volume name>\\_data`.

2. Run a container that mounts this volume, and list the filesystem therein:

```
PS: node-0 Administrator> docker container run `
       -it -v demovol:C:\demo `
       mcr.microsoft.com/windows/servercore:10.0.17763.805 cmd

C:\>dir
 Volume in drive C has no label.
```



```
 Volume Serial Number is 38CD-4889

 Directory of C:\

10/30/2019  05:45 PM    <DIR>          demo
09/15/2018  09:42 AM             5,510 License.txt
10/06/2019  10:04 AM    <DIR>          Program Files
10/06/2019  10:02 AM    <DIR>          Program Files (x86)
10/06/2019  10:05 AM    <DIR>          Users
10/30/2019  05:46 PM    <DIR>          Windows
               1 File(s)          5,510 bytes
               5 Dir(s)  21,209,698,304 bytes free
```

The `demo` directory is created as the mountpoint for our volume, as specified in the flag `-v demovol:C:\demo`.

3. Put some text in a file in this volume; this is analogous to your containerized application writing data out to its filesystem:

```
C:\>cd demo

C:\demo>dir > data.dat
```

4. Exit the container, and list the contents of your volume on the host:

```
PS: node-0 Administrator> ls C:\\ProgramData\\docker\\volumes\\demovol\\_data
```

You'll see your `data.dat` file present at this point in the host's filesystem. Delete the container:

```
PS: node-0 Administrator> docker container rm -f <container ID>
```

The volume and its contents will still be present on the host.

5. Start a new container mounting the same volume, and show that the old data is present in your new container:

```
PS: node-0 Administrator> docker container run `
       -it -v demovol:C:\demo `
       mcr.microsoft.com/windows/servercore:10.0.17763.805 cmd

C:\>cd demo

C:\demo>dir
 Volume in drive C has no label.
 Volume Serial Number is 38CD-4889

 Directory of C:\demo

10/30/2019  05:45 PM    <DIR>          .
10/30/2019  05:45 PM    <DIR>          ..
10/30/2019  05:47 PM               330 data.dat
               1 File(s)            330 bytes
               2 Dir(s)  36,361,719,808 bytes free
```

`data.dat` is recovered from the volume in this new container.

6. Exit this container, and inspect its mount metadata:

```
PS: node-0 Administrator> docker container inspect <container ID>

    "Mounts": [
        {
            "Type": "volume",
            "Name": "demovol",
            "Source": "C:\\ProgramData\\docker\\volumes\\demovol\\_data",
            "Destination": "c:\\demo",
```



```
            "Driver": "local",
            "Mode": "",
            "RW": true,
            "Propagation": ""
        }
    ],
```

Here we can see the volumes and host mountpoints for everything mounted into this container.

7. Clean up by removing that volume:

   ```
   PS: node-0 Administrator> docker volume rm demovol
   ```

   You will get an error saying the volume is in use - docker will not delete a volume mounted to any container (even a stopped container) in this way. Remove the offending container first, then remove the volume again.

3.2 Mounting Host Paths

1. In a fresh directory `myweb`, make a Dockerfile to make a simple containerization of nginx:

   ```
   FROM mcr.microsoft.com/windows/servercore:10.0.17763.805
   RUN ["powershell", "wget", "http://nginx.org/download/nginx-1.11.6.zip", \
        "-UseBasicParsing", "-OutFile", "c:\\nginx.zip"]
   RUN ["powershell", "Expand-Archive", "c:\\nginx.zip", "-Dest", "c:\\nginx"]
   WORKDIR c:\\nginx\\nginx-1.11.6
   ENTRYPOINT ["powershell", ".\\nginx.exe"]
   ```

2. Build this image, and use it to start a container that serves the default nginx landing page:

   ```
   PS: node-0 myweb> docker image build -t nginx .
   PS: node-0 myweb> docker container run -d -p 5000:80 nginx
   ```

   Visit the landing page at `<node-0 public IP>:5000` to confirm everything is working, then remove this container.

3. Create some custom HTML for your new website:

   ```
   PS: node-0 myweb> echo "<h1>Hello Wrld</h1>" > index.html
   ```

4. The HTML served by nginx is found in the container's filesystem at `C:\nginx\nginx-1.11.6\html`. Mount your `myweb` directory at this path:

   ```
   PS: node-0 myweb> docker container run -d -p 5000:80 `
       -v C:\Users\Administrator\myweb:C:\nginx\nginx-1.11.6\html `
       nginx
   ```

   Visit your webpage `<node 0 public IP>:5000`; you should be able to see your custom webpage.

5. There's a typo in your custom html. Fix the spelling of 'world' in your HTML, and refresh the webpage; the content served by nginx gets updated without having to restart or replace the nginx container.

3.3 Conclusion

In this demo, we saw two key points about volumes: first, they persist and provision files beyond the lifecycle of any individual container. Second, we saw that manipulating files on the host that have been mounted into a container immediately propagates those changes to the running container; this is a popular technique for developers who containerize their running environment, and mount in their in-development code so they can edit their code using the tools on their host machine that they are familiar with, and have those changes immediately available inside a running container without having to restart or rebuild anything.

4 Instructor Demo: Single Host Networks

In this demo, we'll illustrate:

- The networking stack created for the default Docker `nat` network
- Attaching containers to docker networks
- Inspecting networking metadata
- How network adapters appear in different network namespaces

4.1 Following Default Docker Networking

1. On a fresh node you haven't run any containers on yet, list your networks:

   ```
   PS: node-1 Administrator> docker network ls

   NETWORK ID          NAME                DRIVER              SCOPE
   03f6ddacab50        nat                 nat                 local
   b0de36ba94f3        none                null                local
   ```

2. Get some metadata about the `nat` network, which is the default network containers attach to when doing `docker container run`:

   ```
   PS: node-1 Administrator> docker network inspect nat
   ```

   Note the `containers` key:

   ```
   "Containers": {}
   ```

   So far, no containers have been plugged into this network.

3. Create a container attached to your nat network:

   ```
   PS: node-1 Administrator> docker container run --name=c1 -dt `
       mcr.microsoft.com/windows/nanoserver:10.0.17763.737
   ```

   The `network inspect` command above will now show this container plugged into the `nat` network, which is the default network containers are attached to if they aren't created with a `--network` key.

4. Have a look at the network adapters created inside this container's network namespace:

   ```
   PS: node-1 Administrator> docker container exec c1 ipconfig /all

   Windows IP Configuration

      Host Name . . . . . . . . . . . . : b201969c45d5
      Primary Dns Suffix  . . . . . . . :
      Node Type . . . . . . . . . . . . : Hybrid
      IP Routing Enabled. . . . . . . . : No
      WINS Proxy Enabled. . . . . . . . : No
      DNS Suffix Search List. . . . . . : us-east-2.compute.internal

   Ethernet adapter vEthernet (Ethernet) 4:

      Connection-specific DNS Suffix  . : us-east-2.compute.internal
      Description . . . . . . . . . . . : Hyper-V Virtual Ethernet Adapter #6
      Physical Address. . . . . . . . . : 00-15-5D-74-66-CC
      DHCP Enabled. . . . . . . . . . . : No
      Autoconfiguration Enabled . . . . : Yes
      Link-local IPv6 Address . . . . . : fe80::81f:26c5:f9a3:b2cf%22(Preferred)
      IPv4 Address. . . . . . . . . . . : 172.17.168.239(Preferred)
      Subnet Mask . . . . . . . . . . . : 255.255.240.0
      Default Gateway . . . . . . . . . : 172.17.160.1
   ```



```
   DNS Servers . . . . . . . . . . . : 172.17.160.1
                                       172.31.0.2
   NetBIOS over Tcpip. . . . . . . . : Disabled
   Connection-specific DNS Suffix Search List :
                                       us-east-2.compute.internal
```

Note the Host Name matches the container ID by default, and the IPv4 address of the virtual ethernet adapter inside the container matches the container IP.

5. Create another container, and ping one from the other by container name:

```
PS: node-1 Administrator> docker container run --name=c2 -dt `
                            mcr.microsoft.com/windows/nanoserver:10.0.17763.737
PS: node-1 Administrator> docker container exec c1 ping c2

Pinging c2 [172.20.134.196] with 32 bytes of data:
Reply from 172.20.134.196: bytes=32 time<1ms TTL=128
Reply from 172.20.134.196: bytes=32 time<1ms TTL=128
Reply from 172.20.134.196: bytes=32 time<1ms TTL=128
Reply from 172.20.134.196: bytes=32 time<1ms TTL=128

Ping statistics for 172.20.134.196:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms
```

The ping is successful; Docker uses DNS resolution so that our application logic (`ping c2` in this case) doesn't need to do any explicit service discovery or networking lookups by hand; all that is provided by the Docker engine and Windows networking stack.

6. Create one final container, but don't name it this time, and attempt to ping it from c1 like above:

```
PS: node-1 Administrator> docker container run -dt `
    mcr.microsoft.com/windows/nanoserver:10.0.17763.737

PS: node-1 Administrator> docker container exec c1 ping <new container name>

Ping request could not find host <new container name>.
    Please check the name and try again.
```

Docker only provides DNS lookup for containers explicitly named with the `--name` flag.

4.2 Forwarding a Host Port to a Container

1. Start an `nginx` container with a port exposure:

```
PS: node-1 Administrator> docker container run -d -p 5000:80 --name proxy nginx
```

This syntax asks docker to forward all traffic arriving on port 5000 of the host's network namespace to port 80 of the container's network namespace. Visit the `nginx` landing page at `<node-1 public IP>:5000` in a browser.

2. Delete all you containers on this node to clean up:

```
PS: node-1 Administrator> docker container rm -f $(docker container ls -aq)
```

4.3 Conclusion

In this demo, we stepped through the basic behavior of docker software defined nat networks. By default, all containers started on a host without any explicit networking configuration will be able to communicate across Docker's `nat`

network, and in order for containers to resolve each other's name by DNS, they must also be explicitly named upon creation.

5 Instructor Demo: Self-Healing Swarm

In this demo, we'll illustrate:

- Setting up a swarm
- How swarm makes basic scheduling decisions
- Actions swarm takes to self-heal a docker service

5.1 Setting Up a Swarm

1. Start by making sure no containers are running on any of your nodes:

```
PS: node-0 Administrator> docker container rm -f $(docker container ls -aq)
PS: node-1 Administrator> docker container rm -f $(docker container ls -aq)
PS: node-2 Administrator> docker container rm -f $(docker container ls -aq)
PS: node-3 Administrator> docker container rm -f $(docker container ls -aq)
```

2. Initialize a swarm on `node-0`:

```
PS: node-0 Administrator> $PRIVATEIP = '<node-0 private IP>'
PS: node-0 Administrator> docker swarm init `
    --advertise-addr ${PRIVATEIP} `
    --listen-addr ${PRIVATEIP}:2377

Swarm initialized: current node (xyz) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-0s96... 10.10.1.40:2377

To add a manager to this swarm, run 'docker swarm join-token manager'
    and follow the instructions.
```

3. List the nodes in your swarm:

```
PS: node-0 Administrator> docker node ls

ID      HOSTNAME   STATUS   AVAILABILITY   MANAGER STATUS
xyz *   node-0     Ready    Active         Leader
```

4. Add some workers to your swarm by cutting and pasting the `docker swarm join...` token Docker provided in step 2 above:

```
PS: node-1 Administrator> docker swarm join --token SWMTKN-1-0s96... 10.10.1.40:2377
PS: node-2 Administrator> docker swarm join --token SWMTKN-1-0s96... 10.10.1.40:2377
PS: node-3 Administrator> docker swarm join --token SWMTKN-1-0s96... 10.10.1.40:2377
```

   Each node should report `This node joined a swarm as a worker.` after joining.

5. Back on `node-0`, list your swarm members again:

```
PS: node-0 Administrator> docker node ls

ID      HOSTNAME   STATUS   AVAILABILITY   MANAGER STATUS
ghi     node-3     Ready    Active
def     node-2     Ready    Active
abc     node-1     Ready    Active
```



```
xyz *   node-0     Ready    Active         Leader
```

You have a four-member swarm, ready to accept workloads.

5.2 Scheduling Workload

1. Create a service on your swarm:

```
PS: node-0 Administrator> docker service create `
    --replicas 4 `
    --name service-demo `
    mcr.microsoft.com/windows/nanoserver:10.0.17763.737 ping -t 8.8.8.8
```

2. List what processes have been started for your service:

```
PS: node-0 Administrator> docker service ps service-demo

ID     NAME             IMAGE       NODE    DESIRED   CURRENT STATE
                                              STATE
g3...  service-demo.1   nanoserver  node-3  Running   Running 18 seconds ago
e7...  service-demo.2   nanoserver  node-0  Running   Running 18 seconds ago
wv...  service-demo.3   nanoserver  node-1  Running   Running 18 seconds ago
ty...  service-demo.4   nanoserver  node-2  Running   Running 18 seconds ago
```

Our service has scheduled four tasks, one on each node in our cluster; by default, swarm tries to spread tasks out evenly across hosts, but much more sophisticated scheduling controls are also available.

5.3 Maintaining Desired State

1. Connect to node-1, and list the containers running there:

```
PS: node-1 Administrator> docker container ls

CONTAINER ID  IMAGE          COMMAND                    ...  NAMES
5b5f77c67eff  54.152.61.101  "powershell ping 8.8.8.8"  ...  service-demo.3.wv0cul...
```

Note the container's name indicates the service it belongs to.

2. Let's simulate a container crash, by killing off this container:

```
PS: node-1 Administrator> docker container rm -f <container ID>
```

Back on our swarm manager node-0, list the processes running for our service-demo service again:

```
PS: node-0 Administrator> docker service ps service-demo

... NAME               IMAGE        NODE    DESIRED   CURRENT STATE
                                             STATE
... service-demo.1     nanoserver   node-3  Running   Running 6 minutes ago
... service-demo.2     nanoserver   node-0  Running   Running 6 minutes ago
... service-demo.3     nanoserver   node-1  Running   Running 3 seconds ago
... \_ service-demo.3  nanoserver   node-1  Shutdown  Failed 3 seconds ago
... service-demo.4     nanoserver   node-2  Running   Running 6 minutes ago
```

Swarm has automatically started a replacement container for the one you killed on node-1. Go back over to node-1, and do docker container ls again; you'll see a new container for this service up and running.

3. Next, let's simulate a complete node failure by rebooting one of our nodes; on node-3, navigate **Start menu -> Power -> Restart**.

4. Back on your swarm manager, check your service containers again; after a few moments, you should see something like:


```
PS: node-0 Administrator> docker service ps service-demo
NAME                IMAGE       NODE    DESIRED    CURRENT STATE
                                          STATE
service-demo.1      nanoserver  node-0  Running    Running 19 seconds ago
\_ service-demo.1   nanoserver  node-3  Shutdown   Running 38 seconds ago
service-demo.2      nanoserver  node-0  Running    Running 12 minutes ago
service-demo.3      nanoserver  node-1  Running    Running 5 minutes ago
\_ service-demo.3   nanoserver  node-1  Shutdown   Failed 5 minutes ago
service-demo.4      nanoserver  node-2  Running    Running 12 minutes ago
```

The process on node-3 has been scheduled for `SHUTDOWN` when the swarm manager lost connection to that node, and meanwhile the workload has been rescheduled onto node-0 in this case. When node-3 comes back up and rejoins the swarm, its container will be confirmed to be in the `SHUTDOWN` state, and reconciliation is complete.

5. Remove your `service-demo`:

   ```
   PS: node-0 Administrator> docker service rm service-demo
   ```

   All tasks and containers will be removed.

5.4 Conclusion

One of the great advantages of the portability of containers is that we can imagine orchestrators like Swarm which can schedule and re-schedule workloads across an entire datacenter, such that if a given node fails, all its workload can be automatically moved to another host with available resources. In the above example, we saw the most basic examples of this 'reconciliation loop' that swarm provides: the swarm manager is constantly monitoring all the containers it has scheduled, and replaces them if they fail or their hosts become unreachable, completely automatically.

6 Instructor Demo: Kubernetes Basics

In this demo, we'll illustrate:

- Setting up a Kubernetes cluster with one master and two nodes
- Scheduling a pod, including the effect of taints on scheduling
- Namespaces shared by containers in a pod

Note: At the moment, support for Kubernetes nodes on Windows is still developing. In these exercises, we'll explore Kubernetes on linux to get familiar with Kubernetes objects and scheduling, which will work very similarly on Windows. Furthermore, Kubernetes masters will for the near future remain linux-only; stay tuned to updates in Docker's Universal Control Plane for upcoming tools for bootstrapping mixed linux / windows Kubernetes clusters.

6.1 Connecting to Linux Nodes

The easiest way to connect to your Linux nodes will be via PuTTY, which is already installed on your Windows nodes (see the desktop shortcut). Provide the public IP for `kube-0`, hit *Open*, and provide the username and password that your instructor will provide you.

6.2 Initializing Kubernetes

1. On `kube-0`, initialize the cluster with `kubeadm`:

   ```
   [centos@kube-0 ~]$ sudo kubeadm init --pod-network-cidr=192.168.0.0/16 \
       --ignore-preflight-errors=SystemVerification
   ```

   If successful, the output will end with a join command:


```
...
You can now join any number of machines by running the following on each node
as root:

  kubeadm join 10.10.29.54:6443 --token xxx --discovery-token-ca-cert-hash sha256:yyy
```

2. To start using you cluster, you need to run:

```
[centos@kube-0 ~]$ mkdir -p $HOME/.kube
[centos@kube-0 ~]$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
[centos@kube-0 ~]$ sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

3. List all your nodes in the cluster:

```
[centos@kube-0 ~]$ kubectl get nodes
```

Which should output something like:

```
NAME      STATUS     ROLES     AGE       VERSION
kube-0    NotReady   master    2h        v1.11.1
```

The `NotReady` status indicates that we must install a network for our cluster.

4. Let's install the Calico network driver:

```
[centos@kube-0 ~]$ kubectl apply -f https://bit.ly/2v9yaaV
```

5. After a moment, if we list nodes again, ours should be ready:

```
[centos@kube-0 ~]$ kubectl get nodes -w
NAME      STATUS     ROLES     AGE       VERSION
kube-0    NotReady   master    1m        v1.11.1
kube-0    NotReady   master    1m        v1.11.1
kube-0    NotReady   master    1m        v1.11.1
kube-0    Ready     master    2m        v1.11.1
kube-0    Ready     master    2m        v1.11.1
```

6.3 Exploring Kubernetes Scheduling

1. Let's create a `demo-pod.yaml` file on `kube-0` after enabling Kubernetes on this single node. Use `nano demo-pod.yaml` to create the file:

```
apiVersion: v1
kind: Pod
metadata:
  name: demo-pod
spec:
  volumes:
  - name: shared-data
    emptyDir: {}
  containers:
  - name: nginx
    image: nginx
  - name: mydemo
    image: centos:7
    command: ["ping", "8.8.8.8"]
```

When done, save with `CTRL+o ENTER`, and quit with `CTRL+x`.

2. Deploy the pod:


```
[centos@kube-0 ~]$ kubectl create -f demo-pod.yaml
```

3. Check to see if the pod is running:

```
[centos@kube-0 ~]$ kubectl get pod demo-pod

NAME       READY     STATUS    RESTARTS   AGE
demo-pod   0/2       Pending   0          7s
```

   The status should be stuck in pending. Why is that?

4. Let's attempt to troubleshoot by obtaining some information about the pod:

```
[centos@kube-0 ~]$ kubectl describe pod demo-pod
```

   In the bottom section titled `Events:`, we should see something like this:

```
...
Events:
  Type     Reason            ...  Message
  ----     ------            ...  -------
  Warning  FailedScheduling  ...  0/1 nodes are available: 1 node(s)
                                  had taints that the pod didn't tolerate.
```

   Note how it states that the one node in your cluster has a taint, which is Kubernetes's way of saying there's a reason you might not want to schedule pods there.

5. Get some state and config information about your single kubernetes node:

```
[centos@kube-0 ~]$ kubectl describe nodes
```

   If we scroll a little, we should see a field titled `Taints`, and it should say something like:

```
Taints:  node-role.kubernetes.io/master:NoSchedule
```

   By default, Kubernetes masters carry a taint that disallows scheduling pods on them. While this can be overridden, it is best practice to not allow pods to get scheduled on master nodes, in order to ensure the stability of your cluster.

6. Execute the join command you found above when initializing Kubernetes on `kube-1` (you'll need to add `sudo` to the start, and `--ignore-preflight-errors=SystemVerification` to the end), and then check the status back on `kube-0`:

```
[centos@kube-1 ~]$ sudo kubeadm join...--ignore-preflight-errors=SystemVerification
[centos@kube-0 ~]$ kubectl get nodes
```

   After a few moments, there should be two nodes listed - all with the `Ready` status.

7. Let's see what system pods are running on our cluster:

```
[centos@kube-0 ~]$ kubectl get pods -n kube-system
```

   which results in something similar to this:

```
NAME                                    READY     STATUS    RESTARTS   AGE
calico-etcd-69x56                       1/1       Running   0          5m
calico-kube-controllers-559c657d6d-2nx8f   1/1       Running   0          5m
calico-node-dwl9v                       2/2       Running   0          5m
calico-node-gv9mt                       2/2       Running   0          58s
coredns-78fcdf6894-44tfj                1/1       Running   0          56m
coredns-78fcdf6894-w97xx                1/1       Running   0          56m
etcd-kube-0                             1/1       Running   0          55m
kube-apiserver-kube-0                   1/1       Running   0          55m
kube-controller-manager-kube-0          1/1       Running   0          55m
kube-proxy-c7wpf                        1/1       Running   0          58s
```



```
kube-proxy-nnsj6                    1/1       Running   0          56m
kube-scheduler-kube-0               1/1       Running   0          55m
```

We can see the pods running on the master: etcd, api-server, controller manager and scheduler, as well as calico and DNS infrastructure pods deployed when we installed calico.

8. Finally, let's check the status of our demo pod now:

```
[centos@kube-0 ~]$ kubectl get pod demo-pod
```

Everything should be working correctly with 2/2 containers in the pod running, now that there is an un-tainted node for the pod to get scheduled on.

6.4 Exploring Containers in a Pod

1. Let's interact with the centos container running in demo-pod by getting a shell in it:

```
[centos@kube-0 ~]$ kubectl exec -it -c mydemo demo-pod -- /bin/bash
```

Try listing the processes in this container:

```
[root@demo-pod /]# ps -aux
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root          1  0.0  0.0  24860  1992 ?        Ss   14:48   0:00 ping 8.8.8.8
root          5  0.0  0.0  11832  3036 pts/0    Ss   14:48   0:00 /bin/bash
root         20  0.0  0.0  51720  3508 pts/0    R+   14:48   0:00 ps -aux
```

We can see the ping process we containerized in our yaml file running as PID 1 inside this container - this is characteristic of PID kernel namespaces on linux.

2. Try reaching Nginx:

```
[root@demo-pod /]# curl localhost:80
```

You should see the HTML for the default nginx landing page. Notice the difference here from a regular container; we were able to reach our nginx deployment from our centos container on a port on localhost. The nginx and centos containers share a network namespace and therefore all their ports, since they are part of the same pod.

6.5 Conclusion

In this demo, we saw two scheduling innovations Kubernetes offers: taints, which provide 'anti-affinity', or reasons not to schedule a pod on a given node; and pods, which are groups of containers that are always scheduled on the same node, and share network, IPC and hostname namespaces. These are both examples of Kubernetes's highly expressive scheduling, and are both difficult to reproduce with the simpler scheduling offered by Swarm.

7 Instructor Demo: Docker Compose

In this demo, we'll illustrate:

- Starting an app defined in a docker compose file
- Inter-service communication using DNS resolution of service names

7.1 Exploring the Compose File

1. Please download the DockerCoins app from Github and change directory to ~/orchestration-workshop-net/dockercoins:


```
PS: node-0 Administrator> git clone -b ee3.0-ws19 `
    https://github.com/docker-training/orchestration-workshop-net.git
PS: node-0 Administrator> cd ~/orchestration-workshop-net
```

2. Let's take a quick look at our Compose file for Dockercoins:

```
version: "3.1"

services:
  rng:
    image: training/dc_rng:ws19
    networks:
    - nat
    ports:
    - "8001:80"

  hasher:
    image: training/dc_hasher:ws19
    networks:
    - nat
    ports:
    - "8002:80"

  webui:
    image: training/dc_webui:ws19
    networks:
    - nat
    ports:
    - "8000:80"

  redis:
    image: training/dc_redis:ws19
    networks:
    - nat

  worker:
    image: training/dc_worker:ws19
    networks:
    - nat

networks:
  nat:
    external: true
```

   This Compose file contains 5 services, and a pointer to the default `nat` network. The images `training/dc_rng:ws19` et cetera are pre-built images containing the application logic you can explore in the subfolders of `~/orchestration-workshop-net`.

3. Start the app in the background:

   ```
   PS: node-0 orchestration-workshop-net> docker-compose up -d
   ```

4. Make sure the services are up and running, and all the containers are attached to the local `nat` network:

   ```
   PS: node-0 orchestration-workshop-net> docker-compose ps
   PS: node-0 orchestration-workshop-net> docker network inspect nat
   ```

5. If everything is up, visit your app at `<node-0 public IP>:8000` to see Dockercoins in action.

7.2 Communicating Between Containers

1. In this section, we'll demonstrate that containers created as part of a service in a Compose file are able to communicate with containers belonging to other services using just their service names. Let's start by listing our DockerCoins containers:

   ```
   PS: node-0 orchestration-workshop-net> docker container ls | findstr 'dc'
   ```

2. Now, connect into one container; let's pick `webui`:

   ```
   PS: node-0 orchestration-workshop-net> docker container exec `
       -it <Container ID> powershell
   ```

3. From within the container, ping `rng` by name:

   ```
   PS C:\> ping rng
   ```

   Logs should be outputted resembling this:

   ```
   Pinging rng [172.20.137.174] with 32 bytes of data:
   Reply from 172.20.137.174: bytes=32 time<1ms TTL=128
   Reply from 172.20.137.174: bytes=32 time<1ms TTL=128
   Reply from 172.20.137.174: bytes=32 time<1ms TTL=128
   Reply from 172.20.137.174: bytes=32 time<1ms TTL=128

   Ping statistics for 172.20.137.174:
       Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
   Approximate round trip times in milli-seconds:
       Minimum = 0ms, Maximum = 0ms, Average = 0ms
   ```

   DNS lookup for the services in DockerCoins works because they are all attached to the local `nat` network.

4. After exiting this container, let's navigate to the `worker` folder and take a look at the top of `Program.cs`:

   ```
   PS: node-0 orchestration-workshop-net> cd worker
   PS: node-0 worker> cat Program.cs

   using System;
   using System.Net.Http;
   using System.Threading;
   using ServiceStack.Redis;

   public class Program
   {
       private static HttpClient Client = new HttpClient();
       private const string rng_uri = "http://rng";
       private const string hasher_uri = "http://hasher";
   ...
   ```

   Our worker is configured to contact the random number generator and hasher directly by their service names, `rng` and `hasher`. No service discovery or IP lookups required - Docker ensures that service names are DNS-resolvable, abstracting away our service-to-service communication.

5. Shut down Dockercoins and clean up its resources:

   ```
   PS: node-0 orchestration-workshop-net> docker-compose down
   ```

7.3 Conclusion

In this exercise, we stood up an application using Docker Compose. The most important new idea here is the notion of Docker Services, which are collections of identically configured containers. Docker Service names are resolvable by DNS, so that we can write application logic designed to communicate service to service; all service discovery and load balancing between your application's services is abstracted away and handled by Docker.

