Miscellaneous commands

Logs

The logs commands is pretty self explanatory; it allows you to interact with the stdout stream of your containers, which Docker is keeping track of in the background. For example, to view the last entries written to stdout for our nginx-test container, I just need to use the following command:

```
$ docker container logs --tail 5 nginx-test
```

To view the logs in real time, I simply need to run the following:

```
$ docker container logs -f nginx-test
```

The -f flag is shorthand for --follow . I can also, say, view everything that has been logged since 15:00 today by running the following:

```
$ docker container logs --since 2018-02-24T16:00 nginx-test
```

top

The top command is quite a simple one; it lists the processes running within the container you specify:

```
$ docker container top nginx-test
```

stats

The stats command provides real-time information on either the specified container or, if you don't pass a container name or ID, all running containers:

```
$ docker container stats nginx-test
```

We are given information on the CPU , RAM , NETWORK , and DISK IO for the specified container. We can also pass the -a flag; this is short for --all and displays all containers, running or not. However, if the container isn't running, there aren't any resources being utilized, so it doesn't really add any value other than giving you a visual representation of how many containers you have running and where the resources are being used.

It is also worth pointing out that the information displayed by the stats command is realtime only; Docker does not record the resource utilization and make it available in the same way that the logs command does.

Resource limits

The last command we ran showed us the resource utilization of our containers; by default, a container when launched will be allowed to consume all the available resources on the host machine if it requires it. We can put caps on the resources our containers can consume; let's start by updating the resource allowances of our nginx-test container.

Typically, we would have set the limits when we launched our container using the run command; for example, to use 2 CPUs and set a memory limit of 128M , we would have used the following command:

```
$ docker container run -d --name nginx-test --cpus=2 --memory 128M -p 8080:80 nginx
```

However, we didn't need to update our already running container; to do this, we can use the update command. Now you must have thought that this should entail just running the following command:

$ docker container update --cpus=2 --memory 128M nginx-test

Running the preceding command will actually produce an error:

Error response from daemon: Cannot update container e6ac3ce1418d520233a68f71a1e5cb38b1ab0aafab5fa00d6e0220975706b246: Memory limit should be smaller than already set memoryswap limit, update the memoryswap at the same time

So what is the memoryswap limit currently set to? To find this out, we can use the inspect command to display all of the configuration data for our running container; just run the following:

$ docker container inspect nginx-test

As you can see, there is a lot of configuration data. When I ran the command, a 199-line JSON array was returned. Let's use the grep command to filter out just the lines that contain the word memory :

$ docker container inspect nginx-test | grep -i memory

This returns the following configuration data:
"Memory": 0,
"KernelMemory": 0,
"MemoryReservation": 0,
"MemorySwap": 0,
"MemorySwappiness": -1,

Everything is set to 0 : how can 128M be smaller than 0 ? In the context of the configuration of the resources, 0 is actually the default value and means that there are no limits--notice the lack of M at the end. This means that our update command should actually read the following:

$ docker container update --cpus=2 --memory 128M --memory-swap 256M nginx-test

Paging is a memory-management scheme in which the kernel stores and retrieves, or swaps, data from secondary storage for use in main memory. This allows processes to exceed the size of available physical memory.

By default, when you set --memory as part of the run command, Docker will set the --memory-swap size to be twice of that of --memory .
If you run docker container stats nginx-test now, you should see our limits in place.

Also, rerunning docker container inspect nginx-test | grep -i memory will
show the changes:

"Memory": 134217728,
"KernelMemory": 0,
"MemoryReservation": 0,
"MemorySwap": 268435456,
"MemorySwappiness": -1,

The values when running docker container inspect are all shown in bytes.

States commands

we are going to look at the various states your containers could be in and the few remaining commands we have yet to cover as part of the docker container command.

Before we continue, let's launch five more containers. To do this quickly, run the following command:

```
$ for i in {1..5}; do docker container run -d --name nginx$(printf "$i")
nginx; done
```

When running docker container ls -a , you should see your five new containers, named nginx1 through to nginx5.

Pause and unpause
Let's look at pausing nginx1 . To do this, simply run the following:

```
$ docker container pause nginx1
```

Running docker container ls will show that the container has a status of Up , but Paused.

As you will have probably already guessed, you can resume a paused container using the unpause command:

```
$ docker container unpause nginx1
```

This command is useful if you need to freeze the state of a container; maybe one of your containers is going haywire and you need to do some investigation later but don't want it to have a negative impact on your other running containers.

Stop, start, restart, and kill
Next up, we have the stop , start , restart , and kill commands. We have already used the start command to resume a container with a status of Exited . The stop command works in exactly the same way as when we used Ctrl + C to detach from your container running in the foreground. Run the following:

```
$ docker container stop nginx2
```

With this, a request is sent to the process for it to terminate, called a SIGTERM . If the process has not terminated itself within a grace period, then a kill signal, called a SIGKILL , is sent.
This will immediately terminate the process, not giving it anytime to finish whatever is causing the delay, for example, committing the results of a database query to disk. Because this could be bad, Docker has given you the option of overriding the default grace period, which is 10 seconds, by using the -t flag; this is short for --time . For example, running the following command will wait up to 60 seconds before sending a SIGKILL , should it need to be sent to kill the process:

```
$ docker container stop -t 60 nginx3
```

The start command, as we have already experienced, will start the process back up; however, unlike

the pause and unpause commands, the process in this case starts from scratch rather than starting from where it left off.

$ docker container start nginx2 nginx3

The restart command is a combination of the following two commands; it stops and then starts the container ID or name you pass it. Also, like stop , you can pass the -t flag:

$ docker container restart -t 60 nginx4

Finally, you also have the option sending a SIGKILL immediately to the container by running the kill command:

$ docker container kill nginx5